# Intelligent Data-Driven Reverse Engineering of Software Design Patterns


**Sultan Ibrahim Alhusain**

Faculty of Technology

De Montfort University

A thesis submitted in partial fulfilment of the requirements

for the degree of

*Doctor of Philosophy*

March, 2016

# Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this or any other university.

Sultan Ibrahim Alhusain

March, 2016

I dedicate this thesis to my parents, wife and son...

*"Far better an approximate answer to the right question, which is often vague, than an exact answer to the wrong question, which can always be made precise".*

**John Tukey**

# Abstract

Recognising implemented instances of Design Patterns (DPs) in software design discloses and recovers a wealth of information about the intention of the original designers and the rationale of their design decisions. Because it is often the case that the documentation available for software systems, if any, is poor and/or obsolete, recovering such information can be of great help and importance for maintenance tasks. Since DPs are abstractly and vaguely defined, a set of software classes with exactly the same relationships as expected for a DP instance may actually be only accidentally similar. On the other hand, a set of classes with relationships that are to an extent different from the typically expected can still be a true DP instance. The deciding factor is mainly whether or not the set of classes is actually *intended* to solve the design problem addressed by the DP, which makes the *intent* a fundamental and defining characteristic of DPs.

Discerning the intent of potential instances requires building complex models that cannot be built using the information known about DPs. So, a paradigm shift in DP recognition to fully machine learning based approaches is required. The problem is that there exists no accurate and sufficiently large DP datasets and it is difficult to manually construct one. Also, there is a lack of research on the feature set that should be used in DP recognition. The main aim of this thesis is to enable the paradigm shift by laying down an accurate, comprehensive and information-rich foundation of feature and data sets. To achieve this aim, a large set of features is developed to cover a wide range of design aspects, with a particular focus on the design *intent*. This set serves as a global feature set from which different subsets can be objectively selected for different DPs. A new and feasible approach for DP dataset construction is designed and used to construct training datasets. The feature and data sets are then used experimentally to build and train DP classifiers. The results demonstrate the accuracy and utility of the sets introduced, and show that fully machine learning based approaches do provide the appropriate and well-equipped solutions to the problem of DP recognition.

## Publications

- Alhusain, S., Coupland, S., John, R., and Kavanagh, M. (2013). Towards machine learning based design pattern recognition. In Proceedings of the 13th UK Workshop on Computational Intelligence (UKCI), pages 244 – 251.
- Alhusain, S., Coupland, S., John, R., and Kavanagh, M. (2013). Design Pattern Recognition by Using Adaptive Neuro Fuzzy Inference System. In Proceedings of the IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI), pages 581 – 587.

# Acknowledgements

First and foremost, I praise the almighty God for giving me this opportunity and granting me the capability to proceed successfully.

I would like to express my deepest love and everlasting gratitude to my father, Eng. Ibrahim Alhusain, and my mother, Mrs. Nawal Almarshad, for their sacrifice, support and prayers. I am indebted to both of them. I am also greatly indebted to my beloved wife, Mrs. Lamia Alasaker, for her constant care, encouragement and never-failing patience. Last but not least, the joy my son, Fahad, used to greet me with after stressful days is something I will always remember. Without them all, I would not have been able to make it through the PhD journey.

I would like to express my sincere gratitude and appreciation to my supervisor, Dr. Simon Coupland, for his invaluable support, patience and guidance. I would also like to extend my gratitude and appreciation to Prof. Robert John and Dr. Maria Kavanagh for their help and encouragement when it was most required. A further debt of acknowledgement is owed to Prof. Yann-Gaël Guéhéneuc and Dr. Nikolaos Tsantalis for their kind help and useful comments.

# Table of contents

# List of Abbreviations

| | |
|---|---|
| DP | Design Pattern |
| OO | Object Oriented |
| AI | Artificial Intelligence |
| GoF | Gang-of-Four |
| ASG | Abstract Syntax Graph |
| UML | Unified Modelling Language |
| SVM | Support Vector Machine |
| RBF | Radial Basis Function |
| ANN | Artificial Neural Network |
| EDP | Elemental Design Patterns |
| ARL | Acceptable Risk Level |
| XML | eXtensible Markup Language |
| SM | Skillings–Mack statistical test |

# Chapter 1

# Introduction

As the last original engineer still working on NASA's Voyager probes is retiring this year, NASA looked to hire someone to keep the now-interstellar spacecrafts going. The job involves programming in 60-year-old languages (i.e. FORTRAN and assembly), and the successful candidate will be assigned the task of modifying the software of the spacecrafts in order to reduce the amount of energy consumed by them. This task is needed to prolong their lives as they would, otherwise, run out of power in a decade. Although there will be up to a year of on-the-job training by the original engineer, he himself is working on a 40-year-old memory which may not always be accurate. The old software documentations may not also be so helpful, and the manager for the Voyager program explains why as follows [165]:

> "That's when it's time to turn back to old documents to figure out the logic behind some of the engineering decisions. [However,] it's easy to find the engineering decisions, but harder to find the reasoning. This means combing through secondary documents and correspondence hoping to find the solution, trying to get in another engineer's head".

The problem of getting into the heads of the original engineers to figure out their intents and the rationale of their design decisions, in order to avoid the

inadvertent introduction of bugs, is a common software maintenance problem. However, expert software designers do not usually solve design problems from first principles. They tend, instead, to reuse good solutions that have previously worked for similar problems. Many of these problem-solution pairs have subsequently been documented and they are called Design Patterns (DP). The most known catalogue of DPs is the one presented in [71] which documents 23 OO DPs. Each DP has an illustrative name and describes, in an abstract manner, a recurring design problem and the essence of its best practice solution. If an implemented instance of a known best practice solution is recognised in the design of a software system, a great deal of information is unearthed and disclosed about the intention of the original designers and the rationale of their design decisions. This is the primary aim and purpose of the reverse-engineering of software DPs.

The following section discusses in more detail the benefits of recognising DP instances and what has motivated the research presented in this thesis. Then, the challenges of DP recognition and the main limitations of existing recognition approaches are discussed in Section 1.2. The research objectives are, then, outlined in Section 1.3. Finally, the structure of the thesis are presented in Section 1.4. It is worthwhile noting that the discussion in the following sections, as well as the whole thesis, always assumes the context of OO software design, which is the context of the DPs of interest in this thesis. DPs are, in fact, normally associated with OO design although the basic principle is equally applicable to any paradigm of software design [152]. It should also be noted that all of the DPs mentioned in this thesis (e.g. the Adapter and the Decorator) are presented and described in [71], to which the interested reader is referred.

## 1.1 Motivation

An old and famous piece of advice in computer science is to *never touch a running system*, and the existence of this advice in itself shows how changing a system has traditionally been associated with unexpected consequences and side-effects. These side-effects are usually the results of ill-informed maintenance decisions [123]. While software documentation can help informing such decisions, it is often the case that the documentation is poor, obsolete or even non-existent [75]. Although some UML[1] modelling tools (e.g. Visual Paradigm [162] and Umbrello [88]) can help in producing up-to-date documentations for software systems by reverse engineering their source code into UML models, such tools merely recover design decisions not the rationale that has led to these decisions. Other architecture reconstruction tools (e.g. CodeCrawler [107] and Understand [144]) can also be used to recover different architectural views at different levels of abstraction, which helps in obtaining a broad and detailed understanding of the target software system, but they too suffer from the aforementioned limitation.

Recognising DP instances, on the other hand, reveals the rationale behind the design decisions made in the software classes that make up the recognised instances [13]. The kind of information that can be revealed includes information about the role different classes play within these instances as well as how they should collaborate in order to solve the design problem addressed by the corresponding DP. So, rather than only showing *what* relationships do exist between classes (i.e. design decisions) as it is the case with ordinary reverse-engineering tools, the reason *why* these relationships exist in the first place (i.e. design rationale) becomes clear. Recognising a Composite DP [71] instance, for example, shows that the instance classes are connected in the way they are connected in order to compose objects into a tree structure that represents a part-whole hierarchy. The availability of this

[1] Unified Modeling Language [141].

information is vital for any maintenance task that involves applying changes to this set of classes or to how they are used by other classes, as otherwise the integrity of the instance can be compromised.

The main objective of DP recognition is to help in the process of (re)documenting software systems [16] in order to better inform any later maintenance task of any type (i.e. preventative, corrective, adaptive or perfective maintenance [152]). The information that can be made available by DP recognition can enrich even documentations that are already up-to-date, given that DP instances can be unwittingly produced by good software designers as suggested in [63]. In fact, Prechelt *et al.* have conducted two controlled experiments in order to test the effect of adding explicit DP references to already well commented source code on the performance of maintainers [135]. Their results show that the explicit DP references have led to completing the maintenance tasks faster and with fewer errors. Since it is estimated that about two-thirds of the total software cost go to software maintenance [152] and about 47% of the maintenance cost goes to software comprehension [133], DP recognition can greatly help in this regard and reduce these costs.

Despite the benefits expected from recognising DP instances, the target of most, if not all, existing DP recognition approaches is actually closer to be design decision recovery than it is to be DP recovery. The reason is that they are mainly relying on manually and subjectively specified recognition rules that target the structure of potential instances while this structure can be only accidentally similar. This limitation is explicitly acknowledged in [11, page 189], for example, in which the authors state that the instances recognised for the Adapter DP are "only possibly Adapter instance[s] in the given design". What determines if they are actually Adapter instances or not is their intents. The importance of the *intent* as a defining characteristic of DPs is discussed in more detail in the next section. Recovering the *intent* corresponds to recovering the reasoning and rationale of the original

designers, which is much harder than recovering design decisions, as mentioned earlier. So, instead of tackling this challenge of the problem of DP recognition, the problem definition has been (either implicitly or explicitly) redefined by existing approaches to make it easier to solve.

Discerning the intent of potential DP instances requires complex models that cannot be manually developed based on the abstract descriptions of DPs, which nonetheless is the conventional method employed by existing DP recognition approaches. So, a paradigm shift in the modelling approach is required. Although the paradigm of machine learning [4] provides the algorithms required to build such complex models based on datasets of DP instances, there exists no accurate and sufficiently large dataset and it is difficult to manually construct one. Pettersson *et al.* suggest that locating DP instances manually is beyond the ability of a single research group if a large system or a large number of systems is to be analysed [132]. Costagliola *et al.*, for example, were able to manually analyse only 30% of a 144 class software system and were able to complete the analysis in 3 other smaller systems [41]. Another main obstacle to the paradigm shift is the lack of research in the feature set that should be used in DP recognition [52]. So, even if datasets of DP instances do exist, there is currently no feature set to represent their DP instances as training instances, which then can be used by machine learning algorithms.

This thesis is motivated by the need to enable this paradigm shift by overcoming the obstacles described in the previous paragraph, which will pave the way for more accurate DP recognition tools. This should, in turn, leads to information-rich documentation of software systems and potentially lower maintenance costs.

## 1.2 Limitations of Existing Approaches

One of the main challenges in DP recognition, besides the challenge of discerning the intent of potential instances, is the vague and abstract nature of DPs. Since the design solutions suggested by DPs are described at an abstract level, the details of their implemented instances vary greatly. Moreover, even these abstract descriptions are rarely followed strictly [17]. This results in seemingly endless possible ways of implementing a single DP (i.e. the variants problem). From a forward-engineering perspective, this abstract nature of DPs is desirable and beneficial as it gives them the flexibility required to solve real-world design problems which may be totally different in their specific requirements and details. However, from a reverse engineering perspective, this same nature poses a challenge for any attempt to recognise DP instances in target software systems.

As a consequence and a manifestation of this nature, a set of classes with exactly the same relationships as expected for a DP instance may not actually be a true instance of the DP [17]. On the other hand, another set of classes with relationships that are to an extent different from what may be typically expected can still be a true DP instance. What determines if the former set is a true instance of the DP or just accidentally similar to it, and if the latter set is a true DP instance despite its differences, is not only how their classes are connected and interacting, but also whether or not these sets of classes are actually *intended* to solve the design problem addressed by the DP. The *intent* can, moreover, distinguish instances of similar DPs like the Decorator and the Proxy DPs, which differ mainly on their intents as noted in [71, page 220] and [34, page 275]. This makes the *intent* a fundamental and defining characteristic of DPs. So, any recognition approach that does not take it into account will not, accurately speaking, be a DP recognition approach, but rather an approach that aims to recognise sets of classes that have similar structure and relationships as expected for DPs.

Most of the recognition approaches that do not employ any machine learning or Artificial Intelligence (AI) algorithm in any step of the recognition process (henceforth, non AI-related approaches) do not even attempt to discern the *intent*. This should be expected given the complexity of the *intent* inference task that is probably impossible to be formalised as a set of few manually specified recognition rules, if any such rules can ever be known in the first place. Some of these approaches have either explicitly considered this as a limitation (e.g. [95]) or claimed that the *intent* is not recoverable from the source code (e.g.[137]), and some other have simply ignored the *intent* problem all together (e.g. [170]).

The non AI-related approaches that have attempted to discern the *intent* do so by using too simple and restrictive techniques. In [53] and [91], for example, class names are simply checked for certain keywords that reflect their *intent* (e.g. the classes in a Strategy DP instance should have the keyword 'strategy' in their names), and the use of such simple technique means that only DP instances that are named as expected will have the chance to be recognised. Another simple technique is used in [43], in which the recognition rules defined for three DPs include a subjective threshold rule that targets the *intent*. In [150], the task of capturing the *intent* has been inaccurately reduced to and assumed to require only simple structural and behavioural checks.

Although the reasoning capabilities provided by AI, and in particular machine learning, algorithms can be very helpful in tackling the problem of DP recognition, including the *intent* inference problem, they have not yet been used properly. The limitations of the existing AI-related approaches are summarised as follows:

- The need to discern the *intent* of potential DP instances is not even considered during the design of most AI-related approaches. In some of these approaches, the *intent* is claimed to be an irrecoverable source code aspect.

- The set of features used are generally limited in both their numbers and the information they can potentially convey. Such sets are almost always selected subjectively for each DP to be recognised.
- All the training datasets used are small and/or subjectively labelled. Also, in the case in which the relatively large dataset is used, the *intent* is explicitly excluded from consideration during the process of labelling the dataset.
- In almost all of the cases in which trained classifiers are used, they have been used to either (1) filter the false positives that are recognised by a previous phase or (2) to reduce the search space for a following phase. This means that the core part of the recognition process is not performed by the trained classifiers.

All of these problems and limitations are revisited and expanded upon in the next chapter.

## 1.3 Research Objectives

The main aim of this thesis is to enable a paradigm shift in DP recognition towards fully machine learning based approaches. The foundation formed by the feature and data sets required for this shift has to be an accurate, comprehensive and information-rich foundation, and it has also to be developed in a systematic manner. This aim is fulfilled by the following objectives:

- To develop a new set of information-rich features that covers the structural (e.g. association relationships between classes) and behavioural (e.g. method invocations between classes) aspects of software design. This set will serve as the global feature set from which different subsets can be objectively selected for different DPs.

- To devise a calculation method for the structural and behavioural features that respects the context of the measured entities in a way that helps to better reveal their design *intents*.

- To take advantage of the presumed impact of DPs on software quality by developing a new set of quality metrics that can further enrich the feature set with more and potentially relevant information. This presumed impact will, however, need to be tested on each quality metric before adding it to the feature set. The confounding effects of other factors that may influence the results of the quality impact analysis need to be avoided. The result of this analysis will be a byproduct of this thesis.

- To design and implement a new and feasible approach to constructing large and reasonably accurate DP datasets. Also, the approach should be designed in a way that leads to constructing information-rich datasets in the sense that they contain a wide variety of instances representing a wide range of DP variants.

- To design and implement a DP recognition system based only on classifiers built and trained by using the feature and data sets introduced in this thesis. The evaluation of the accuracy performance of this system will be used to evaluate and demonstrate the adequacy of the feature and data sets introduced. Also, the question of whether or not the quality metrics should be used in DP recognition needs to be answered based on the effect of adding them on the accuracy of the recognition system.

## 1.4 Structure of the Thesis

The remainder of this thesis is organised into the following six chapters:

- **Chapter 2** provides a background information on software DPs and their relationship with software quality. Previous work on DP recognition, as well as previous work on the analysis of quality impact of DPs, are also reviewed and critically discussed. The shortcomings and limitations identified in this chapter inform the research reported in following chapters.

- **Chapter 3** presents a novel set of structural and behavioural features, for which a set of criteria is defined. A context-based normalisation approach to feature value calculation is proposed with the aim of providing more potentially relevant information for the task of discerning the intent.

- **Chapter 4** presents a novel set of quality metrics to be used in the analysis of DP impact on software quality, some of which will also be used as input features in the classifiers trained to implement the recognition system. A novel threshold-based normalisation approach is introduced to account for an identified confounding factor.

- **Chapter 5** presents a novel and feasible approach to DP dataset construction. The approach is implemented to recover DP instances from 539 open source software systems, which makes it the largest and most accurate dataset available for DPs. Besides the accuracy of the datasets, the variety of the DP instances that are represented in the constructed datasets is also discussed and considered during the design and implementation of the approach.

- **Chapter 6** is the chapter in which the contributions of the previous chapters come together and put in use to design and implement a multi-phase DP recognition system. The quality impact of DPs is also evaluated in this chapter.

Two versions of the recognition system, with and without the quality metrics, are developed to study the effect of including them as input features on the recognition accuracy. The accuracy of the trained classifiers will be evaluated and compared with other recognition tools based on an independently peer-reviewed benchmark.

- **Chapter 7** provides the conclusions and summarises the contributions made in this thesis. Some directions and opportunities for future research are also provided.

# Chapter 2

# Background and Related Work

This chapter provides, in Section 2.1, background information on software DPs and their relationship with software quality. Then, in Section 2.2, some of the existing approaches to DP recognition are discussed. The chapter also discusses studies that have attempted to validate the presumed impact of DPs on software quality in Section 2.3. Finally, Section 2.4 presents the outcomes of this chapter and how the work presented here relates to the rest of this thesis.

## 2.1 Software Design Patterns

This section provides a brief introduction to what DPs are and how they have come into being, as well as the main benefits they presumably provide to software quality.

### 2.1.1 Software Patterns

Based on an eight year experience in the field of architecture, Alexander *et al.* have distilled problems that occur repeatedly along with the essence of their solutions. They called these abstract problem-solution pairs *patterns*, in terms of which they have documented their problem solving experience. They suggested that these

patterns can be used to "improve your town and neighborhood" [1, page x]. About ten years later, in 1987, this *pattern* idea was picked up and applied in the field of software engineering [104], which has then been received enthusiastically by the software engineering community driven by an intense desire to improve software quality [33]. The idea has also been described as potentially "one of the most relevant contributions to the field of software engineering in the past 20 years" [49, page 217].

Today, there are catalogues of patterns for all granularity levels of abstraction (i.e architectural, design and implementation) in software systems [34]. There are also catalogues of patterns for the analysis stage of software development [67] as well as domain-specific ones (e.g. networking [143] and game development [26]). However, the seminal and most influential work on software patterns is the book (Design Patterns: Elements of Reusable Object-Oriented Software [71]) authored by four authors who are now known as the Gang-of-Four (GoF). The concept of software patterns was particularly popularised by the GoF book [5], and has consequently revolutionised software design [154].

### 2.1.2 Design Patterns

What is commonly known as software DPs refers to OO software patterns at the design-level of abstraction. This type of software patterns, to which GoF's patterns belong, is the type of interest in this thesis, and so, the discussion will henceforth be limited to this type.

Software DPs, like other types of patterns, are normally described in different catalogues by using a consistent template format that consists of sections describing the design problems to be solved along with the solution to these problems. The description normally includes a discussion of the forces that determine the constraints and the consequences of applying the design solution suggested. Since the

forces can be contradictory, the suggested solutions provide appropriate trade-offs which best serve the intent of the DPs [34]. The Mediator DP, for example, reduces coupling between classes at the cost of having some classes with an increased complexity in order to achieve the goal of having loosely coupled interacting classes, which is stated in the intent section of its description [71].

The design solutions proposed by DPs show how responsibilities should be distributed between software classes and how these classes should collaborate in order to effectively solve the corresponding design problems. The set of responsibilities assigned to a class defines the *role* it plays in solving the design problem. The solution is not, however, described and specified to the finest detail as it is not intended to be a prefabricated solution to be used as is. It is, instead, meant to provide a scheme for a generic solution that can be adopted and customised for the specific design problem at hand [34].

Applying a DP solution to solve a design problem in a software system creates what is called a DP *instance* or *occurrence*[1]. Each instance consists of a number of *participant* classes playing different *roles* within the instance. Roles can, however, be played by (or, in other words, mapped to) multiple classes simultaneously. Also, a single class can play different roles and participate in different DP instances. A class playing the Command role in a Command DP instance, for example, may also play the Component role in a Composite DP instance in cases where a sequence of commands needs to be executed, as explained in [71, page 235]. Different instances of the same DP can vary both at the design level (e.g. adding extra levels of inheritance between two classes playing two directly related roles) as well as at the implementation level (e.g. one-to-many associations can be implemented by an array or a container class), resulting in what is called DP *variants*.

[1] **Note** that the italic terms defined in this paragraph and the preceding one will be used throughout the thesis.

### 2.1.3 Design Patterns and Software Quality

Since DPs are well-proven and best practice solutions that embody collective and cumulative design experience, it is intuitively expected that the proper application of these solutions would lead to better quality software. While experts may inadvertently produce these best practice solutions based on their own experience [63], novices can use DPs as a shortcut to producing such best practice solutions, which may otherwise take them years to accumulate enough experience to produce [71]. Either way, their mere existence in a software design can be used as a measure of the software quality, as suggested by Balanyi and Ferenc [17]. Others have gone a step further by devising a quality evaluation model based on DPs [76].

The reason why DPs are being described as best practices, and consequently assumed to be promoting good quality, is that they themselves are implementations of good OO design principles. One of the most common principles implemented by many DPs is the *encapsulate the aspects that vary* principle. Another important principle is the *open-closed* principle. Both of these two principles help to avoid introducing *bugs* by repeatedly changing existing code [69]. Other design principles include, for example, *program to an interface not an implementation*, which reduces the dependencies between classes, and *favour object composition over class inheritance*, which helps to have a small hierarchy of classes each of which is focused on one task [71].

Different DPs implement different sets of design principles to solve different design problems with different forces and trade-offs. So, it follows naturally that different DPs improve and deteriorate different quality aspects. This has actually been noted in [7], for example. However, most of the DPs in the GoF book "aim at reducing coupling and increasing flexibility within systems" [134, page 1134].

## 2.2 Existing Approaches to DP Recognition

This section discusses existing AI-related approaches to DP recognition. AI-related approaches are the ones that use any AI-related method in any phase of the recognition process. The reason for focusing on this category of approaches is that they are the most closely related to the work presented in this thesis, and they also exemplify and share similar shortcomings and limitations as other approaches.

Each of the following subsections discusses one approach, and the discussions start with brief descriptions of the approaches followed by some critical comments about issues that are particular to them, if any. The subsection titles refer to the names given to the tools in which the approaches are implemented or to the name of the main technique used in the proposed approach. Then, in the last subsection (2.2.10), the main shortcomings and limitations that are common to several approaches are discussed, and a summary of the discussed recognition approaches is provided.

### 2.2.1 FUJABA

A DP recognition approach based on sub-graph isomorphism was proposed in [125] and implemented in FUJABA[2] environment [122]. In this approach, the software system to be analysed is represented as an abstract syntax graph (ASG)[3], and every DP to be recognised is defined based on sub-patterns (i.e. sub-structures that are common to different DPs), which in turn are defined as graph transformation rules. Each DP variant, even if it is only slightly different, needs to have its own set of rules in order to be recognised, which results in a large number of graph transformation rules. The large number of rules intensifies the scalability problem (sub-graph isomorphism is an NP-complete problem), and yet it cannot possibly

[2]FUJABA stands for "From UML to Java And Back Again".
[3]Abstract Syntax Graph (ASG) is a graph representation of the source code of a software system.

guarantee to cover all possible variants. In an attempt to mitigate this problem, similar rules were replaced in a later work [126] by more general and less precise ones, and the resulting uncertainty was handled by associating each rule with a *fuzzy belief* value expressing its probability of producing correct matches. The rules' *fuzzy beliefs* can initially be assigned estimated values, which can later be updated to reflect their true positive ratios [124].

Although this approach is clearly using probability and does not use any fuzzy inference system, it is still claimed to be a fuzzy logic based approach as the title of one of the authors' articles (i.e. [123]) indicates[4]. The concept of *degree of truth* is apparently confused with the concept of *degree of belief*, and such confusion is pervasive even in the literature of AI and uncertainty modelling, as noted in [56]. While the truth in the former is a matter of degree, which is what is meant to be captured by fuzzy logic, it remains binary in the latter[5] (e.g. 50% probability of being correct does not mean 50% correct as the ultimate truth is still either correct or incorrect).

### 2.2.2 Columbus

Balanyi and Ferenc proposed a DP recognition approach in [17], which they have implemented in Columbus reverse engineering framework [64]. In this approach, candidate classes are first identified for each DP role based on the existence of all required attributes, methods and relationships. Then, a detailed check, which includes checking for the presence of associations as well as method delegations, is performed on each possible combination of candidate classes. Each combination that exactly matches the definition of a DP is reported as an instance of that DP.

[4]This claim has, surprisingly, not been questioned in the literature, and the approach was sometimes referred to as if it was indeed a fuzzy logic based (e.g. in [15], [22] and [39]).

[5]In [56, page 45], the standard analogical example of a bottle of liquid is used to explain the difference between the *degree of truth* and the *degree of belief*.

Despite the detailed checks performed, many false positives were recognised. So, in a later work [63], an experiment was reported in which a machine learning based phase was added with the aim of filtering out these false positives. The instances recognised for each DP were manually labelled as true/false positives and used to create a training dataset for the corresponding DP. The dataset of each DP had its own set of subjectively selected input features. The datasets were then used to train the filtering classifiers.

Although the accuracy results of the filtering phase are said to be promising, the accuracy measures used are in fact misleading. While it is reported that, for example, 86.44% of the Adapter false positives were recognised as such, this came at the cost of mistakenly filtering out 20 out of the 25 true positive instances.

### 2.2.3 MARPLE

A DP recognition approach with an integrated machine learning based filtering phase was proposed in [173], and it was implemented in a tool called MARPLE[6] [16]. The recognition process in this tool is performed in two consecutively executed modules: joiner and classifier. In the joiner module, all potential DP instances that satisfy a given rule are recognised by using graph matching techniques. Recognised DP instances that share the same key role-playing classes are then merged into a single instance before being passed into the classifier module. Many false positives are expected to be passed by the joiner module, and the task of the classifier module is to filter them out. The classifier module is implemented by, as the name suggests, classifiers that have been trained by datasets built by manually labelling the DP instances passed by the joiner module. Features that are deemed relevant, over which the recognition rules are defined, are discarded and not fed into the classifiers.

[6]Metrics and ARchitecture Reconstruction PLugin for Eclipse

Merging related instances before the filtering phase may negatively affect the overall accuracy of the recognition tool. This is because in cases where a set of merged instances includes false and true positive instances, filtering the whole merged set will decrease the recall and accepting it will decrease the precision. Another disadvantage of merging related instances, from an instance representation perspective, will be discussed in Chapter 6 (Section 6.1.3). Also, depriving the filtering classifiers from having access to the relevant features may make sense given that they have already been checked. However, such features may provide useful information through their interaction with other supposedly irrelevant features, which may help in improving the classification accuracy.

### 2.2.4 PTIDEJ

The exploratory study[7] reported in [77] was conducted in order to asses the benefits of preceding a DP recognition tool, which implements a constraint satisfaction based recognition approach (i.e. DeMIMA[8] [75]), by a search space reduction phase implemented by a set of data driven rules (i.e. fingerprinting [78]). The rules (or *fingerprints* as it was called) were inferred from a small peer reviewed repository of DP instances, and were expressed as conditions over the values of 13 class-level structural and quality metrics. The rules inferred were then used to reduce the search space by identifying subsets of candidate classes for their corresponding DP roles. The study found the search-space reduction phase to be useful in significantly reducing the time required by DeMIMA while, at the same time, improving its precision without compromising its recall.

[7] PTIDEJ (Pattern Trace Identification, Detection, and Enhancement in Java) is the name of the tool suite used in the study, hence the section title.

[8] Design Motif Identification Multilayered Approach

### 2.2.5 Support Vector Machine (SVM)

Chihada *et al.* have recently proposed a machine learning based approach to DP recognition [39]. The same small peer-reviewed repository as the one used by PTIDEJ was also used here as a training dataset, and a feature set of 45 class-level (structural and quality) metrics were calculated for each role-playing class in every training instance[9]. The order of the role-playing classes within each instance was then permuted to generate more training instances, which were then used to train different classifiers (i.e. different types of SVMs as well as some other classification models). Since the relationships between classes were not represented in the training instances, the set of classes in test systems were first searched for all combinations of classes that are connected by any type of structural relationships (e.g. association), which were then passed to and evaluated by the trained classifiers.

Besides the work presented in this thesis and its published pilot studies (i.e. [2] and [3]), this approach is the only other existing approach that relies on machine learning based models to perform the core recognition task[10]. However, it suffers from three serious methodological problems. First, the permutation used to generate more training instances will in fact corrupt the dataset, as the information provided by individual features and their interactions will be completely lost or, at lease, severely contaminated and distorted. The second problem is the complete disregard of the type and direction of the relationships between classes in potential DP instances. Testing the classification performance of the trained classifiers on systems that are included in the training dataset is the third problem. Similarly to the MARPLE approach discussed above, this approach also merges related instances but by using a different method, which will also be discussed in Chapter 6.

[9]For example, the training instances of a DP with four roles will have a feature vector with 180 features (i.e. $45 \times 4$).

[10]Not search space reduction or filtering false positives out as done by other approaches.

### 2.2.6 Artificial Neural Network (ANN)

Uchiyama *et al.* proposed a DP recognition approach in which the search space is reduced by using a single ANN model that is trained to identify candidate classes for all roles in all DPs [161]. Existing instances of DPs were manually located and collected from several software systems to be used as training instances, each of which was represented by a feature set of 12 class level OO metrics. Candidate classes identified by the trained ANN model are passed to the next phase in which the recognition task is completed by using simple structural matching.

Using a single ANN classifier to identify candidate classes for all DP roles does not allow for the selection of different input features that best suit individual and characteristically different roles. Also, the probability of adding a class playing multiple roles to the candidate lists of all of its roles is likely to decrease. This is because in order for this to happen, all of the relevant normalised neural outputs would need to exceed a given threshold.

### 2.2.7 Fuzzy Matching

A DP recognition approach based on normalised cross-correlation similarity measure was proposed by Wang *et al.* [164]. In this approach, the similarity of each combination of classes with respect to the DP to be recognised is summarised in three similarity scores, each of which represents a design aspect (i.e. class-level, structural and behavioural aspects). The similarity scores are then weighted according to their presumed importance to the DP at hand, and the overall similarity value is calculated as the maximum weighted similarity scores. A DP instance is detected whenever the overall similarity exceeds a threshold. This approach is said to be using fuzzy set theory, and the similarity scores are *apparently* assumed to represent the degree to which a potential instance belongs to the set of true DP instances according to their corresponding aspect.

This approach uses an unusual method of weighting similarity scores (i.e. by using the minimum between a score and its weight[11]) which, together with using the maximum to calculate the overall similarity score, will lead to the recognition of many false positive instances. This is because if, for example, a potential instance happens to be similar to a DP in one and only one design aspect, it can still be reported as a (false) positive instance as long as the similarity score and the weight assigned to this aspect exceed the threshold. This problem could have been avoided by employing a more appropriate evaluation method. The Sugeno fuzzy inference system [121], for example, is a better alternative in which the similarity (i.e. degree of membership) of each aspect is evaluated in a separate rule, and then the overall similarity is calculated by aggregating their membership degrees using the weighted average.

### 2.2.8 Hierarchical Clustering

The DP recognition approach proposed by Czibula and Czibula is based on a hierarchical clustering algorithm [42]. In this approach, the classes of a software system are clustered, and the distance function used for clustering is calculated as the number of unsatisfied DP constraints. The generated clusters can then be reported as instances for a DP, but only if all of their classes satisfy all of the constraints specified for the DP.

There are two main problems with this approach. First, the distance function was defined in a way that makes it symmetric, or otherwise it would be impossible to use it as a distance function; hence the direction of class inter-relationships is disregarded. This may result in splitting the classes of a single DP instance into different clusters. The second problem is that, if a DP instance is assigned to a

[11] The sum of weights was not restricted to equal unity.

cluster which happens to include another non-participant class, the whole cluster, including the DP instance, may be ruled out and discarded.

### 2.2.9 Others

Dong *et al.* claimed that DP instances cannot be represented as normal training instances (i.e. an input feature vector and a target label in each training record) [51]. This claim was based on the assumption that they can only be represented as a combination of interrelated records, each of which represents an individual class. To solve this problem, they proposed a matrix transformation approach to merge and represent DP instances as compound training records. Nevertheless, DP instances can in fact be represented as normal training instances, and a number of existing approaches to do so, as well as the representation approach proposed in this thesis, will be discussed in Chapter 6 (Section 6.1.3).

In [140], a search space reduction phase was proposed in which classes were clustered into different clusters based on their lexical similarity. The idea is to run any DP recognition algorithm on individual clusters separately, which would require less computational time and resources. Two experiments are reported in which two existing DP recognition tools were used with and without reducing the search space, and it was claimed that the proposed phase improved the precision and did not affect the recall. One problem with this approach is its reliance on lexical similarity between the participant classes of individual DP instances, which is based on the assumption that they all implement similar services. However, such classes may implement totally different services as it is the case, for example, with the Receiver classes in the Command DP instance discussed in [69, page 194].

In [119], a set of 40 features were defined for several DP roles and UML stereotypes (e.g. feature counting the number of factory methods[12]). Then, several

[12] As the name suggests, this feature is added for Creator classes in Factory Method DP instances.

classifiers (e.g. Linear Discriminant Analysis) were trained to recognise instances of the roles and the stereotypes by using a small and manually prepared dataset. The trained classifiers were evaluated by performing a cross validation on the training data and the results were described as being satisfactory.

An expert system based approach for DP recognition was proposed in [84]. In this approach, the system under analysis and the DP to be recognised were respectively transformed into a set of facts and a knowledge base of rules. An inference engine was then used to carry out the reasoning required for the recognition task. No experimental results were reported.

### 2.2.10 Discussion

Although the capabilities provided by AI and machine learning models and algorithms can be very helpful in tackling the complexity and ambiguity of the DP recognition problem, including that of discerning the *intent*, they have not yet been used properly. The failure to tackle or consider the *intent* problem, in particular, is the main shortcoming and limitation of existing approaches, and most of the other shortcomings and limitations are, in a way, consequences of this main one. The following subsections discuss these common shortcomings and limitations.

#### 2.2.10.1 Lack of Appropriate Intent Reasoning:

As explained above, this is the main shortcoming and limitation observed in existing DP recognition approaches. In the MARPLE approach [173], despite having an integrated machine learning based filter module, the *intent* was explicitly excluded from consideration during the process of labelling the training dataset as it was claimed to be irrelevant, as will be discussed later. Although the authors of the DeMIMA approach [75] claim that the intent is not an observable characteristic in the source code, the goal of their approach was correctly stated in order to reflect

this limitation. However, in the data-driven search space reduction phase that has been experimentally used with the DeMIMA approach, the goal stated was to "find quantitative signatures common to classes playing given roles" [78]. Such signatures include traces of intent as can be seen in some of the signatures learnt which was found to correspond to the *purpose* of some roles. The authors of the Columbus approach [17] have also suggested that the intent can only be recovered manually. However, in a later work in which they proposed a false-positive filtering phase, they suggested that the complexity of the task of distinguishing false positives from true positives leads naturally to the application of machine learning methods [63]. So, they have defined 6 different predictors for the classifiers trained for two DPs, some of which are explicitly targeting the intent.

Although the authors of the fuzzy matching approach claim that their approach do consider the intent of potential DP instances, they were referring to matching the collaborations between a set of classes with the collaborations expected in a DP instance [164]. If this was indeed what is required to recover and reason about the intent, it will be easily and indisputably recoverable, but it is clearly not. Other approaches have not discussed or even mentioned the problem of recovering the intent.

#### 2.2.10.2 Limited Design Information:

The feature sets, which are either used as predictors in the input feature set of trained classifiers or used as variables over which recognition rules are defined, are generally limited and do not provide much design information. In some approaches (e.g. DeMIMA [75] ), only structural relationship features are used which deprive the recognition analysis from important information regarding how classes in potential DP instances collaborate. So, as it has been observed in [125] and [17], for example, the reliance on structural information only can produce many false

positives. In [39], despite being proposed as DP recognition approach (i.e. not search space reduction approach), only class-level information is provided to the classifiers performing the recognition, and the types and directions of the structural relationships between classes are not checked as long as they are connected.

In approaches that do use behavioural relationship features (e.g. FUJABA [125] and MARPLE [173]), information about method invocations is reduced to binary features representing the existence/absence of method invocations between classes. Although, in an earlier paper that reported the MARPLE approach [13], one of the future works suggested was to "avoid reducing feature values to Booleans", this suggestion was not realised in their later work. Although the same applies to the Columbus recognition approach [17], the filtering phase added in [63] includes predictors that capture, for example, the number of public methods in Adapter candidate classes that invokes a method in their corresponding Adaptee candidates.

The information that is included in the recognition analysis of any approach reflects the goal of the approach. This explains why whenever the goal includes capturing the intent, more information is included by not reducing the feature values into binary ones, as it is the case in the Columbus filtering phase [63]. However, when the goal of the approach is to only recognise a group of classes that are connected by similar relationships as expected for DPs, only binary features are included as they provide sufficient information to achieve this goal.

#### 2.2.10.3 Subjective Selection of Features:

Although finding the best subset of features is critical and crucial for the success of any pattern recognition task [57], there is a lack of research on the set of features that are relevant to different DPs [52]. Consequently, researchers generally resort to subjective selection of features based on their personal views on what is relevant and what is not. In [51], for example, the decision of what features to include comes

down to if the researchers "feel" that they would add relevant information. The only exception is in [119], in which a wrapper based feature selection method was used to objectively select subsets of features for individual DP roles. Feature selection methods have, however, also been used in an earlier version of the MARPLE approach [13], but was not used in their later work.

Besides the fact that these subjective decisions will limit the information available to the recognition analysis to only those that are deemed relevant, these decisions have mostly been taken with only the structural and behavioural characteristics of DPs in mind. For example, in the data-driven search space reduction phase proposed in [78], metrics related to the attributes of classes were not included because they do not correspond to any structural characteristics used to define DPs. Such features may, however, be relevant and helpful in exposing the intent of classes. Either way, the selection of what may and may not be relevant should not be made subjectively.

#### 2.2.10.4 Subjective and Inexpressive Rules

Whenever the recognition rules are not data-driven but rather defined based on the description of DPs, they will be subjective as they would reflect the personal interpretations of DPs and the perceived importance of their characteristics. Subjective rules are currently pervasive throughout the literature of DP recognition, and that is probably why different recognition approaches produce widely disparate results when applied on the same software systems, as observed by [24], [52], [102] and [173].

All of the approaches that are discussed in this chapter use a set of subjective rules to perform the core part of the recognition process. Although the approach proposed in [39] may be considered an exception, it was at the cost of disregarding the types and directions of the inter-class relationships, as mentioned earlier.

Trained classifiers have, however, been used to implement result filters in Columbus [63] and MARPLE [173], and search reduction approaches in [119] and [161], as well as the data-driven search space reduction phase in [78]. Nevertheless, the benefits of these data-driven pre- and post-processing steps are likely to be hindered by the subjective rules that perform the main recognition task. This is because, for example, a DP instance will not have a chance to reach the data-driven filter unless it satisfies the preceding subjective rules.

Besides the subjectivity of non data-driven recognition rules, their limited expressive power can also be a problem. As acknowledged in [77], some DPs are not expressible as a constraint system using the set of constraints defined in DeMIMA. Also, the authors of Columbus stated that all of the instances recognised for the Adapter DP were false positives because they "could not describe the delegation of the important part of the job" [17], which have apparently prompted them to propose a machine learning based phase in [63] to filter the results.

If the importance of DP intents in their recognition had been widely acknowledged and recognised, subjective recognition rules would not have seemed to be a viable solution in the first place. This is because the inclusion of the intent into the set of characteristics to be reasoned about would reveal the real complexity of DP recognition problem in a way that makes any attempt to solve it using a set of few recognition rules seems too simplistic.

#### 2.2.10.5 Subjective Labelling of Datasets:

Most of the datasets used for training have been manually and subjectively labelled. The subjectivity of DP instance evaluation is in fact acknowledged by the authors of the MARPLE approach, for example, who described it as "a task with a high degree of subjectivity" [173]. So, the datasets produced will be representing the concept of the DPs as understood and interpreted by the evaluators but not necessarily as

these concepts objectively are, and consequently, any classifier trained by them is likely to have a high degree of induced subjectivity. This means that, even with the use of trained classifiers, the problem of having subjective recognition rules is still not solved, at least not completely. The same applies for the other approaches that use similarly prepared datasets (e.g. Columbus [63]).

The interpretation of DPs used by MARPLE's authors also appears to be rather loose as they have labelled 1361 DP instances as true positives while only 61 instances are labelled as such in the peer-reviewed P-MARt[13] repository [73] for the same set of DPs. Moreover, the evaluation criteria set for labelling the instances of their dataset are related to structural and behavioural aspects of DP instances, not whether or not these instances have the right intent. In fact, it is stated in [172], in which a wider discussion of the evaluation criteria is reported, that "a correctly implemented pattern instance, but implemented for the wrong reason, is still a correct one"[14], which makes their dataset a dataset of design structures that are similar to DPs. The criteria used to label the other similarly prepared datasets have not been reported.

The peer-reviewed P-MARt repository [73] is the only available dataset that is not labelled subjectively, and it was used to infer search space reduction rules in [78] and to train the classifiers used in [39]. However, besides the methodological problems in the way in which this dataset was used in the latter reference (discussed in Section 2.2.5), this repository contains a relatively small number of DP instances from only 9 software systems. So, any inferred recognition rules will probably not generalise well to the population of software systems.

[13]Pattern-like Micro-Architecture Repository

[14]Their aim of adopting this general labelling rule is to enable the collection of more design information that is "not necessarily related to particularly high-level intent" [172, page 17].

#### 2.2.10.6 Limited Ability to Recognise Variants:

The approach proposed in [164] is an attempt to employ fuzzy set theory to deal with the variant problem, but it fails to use fuzzy logic which is the natural way of reasoning with fuzzy sets. They have, instead, devised their own simplistic and ill-designed inference method that leads to the recognition of potentially many false positives, as explained in Section 2.2.7.

In FUJABA, each DP variant to be recognised has to have its own set of recognition rules, which are defined in an easy-to-modify catalogue [125]. The large number of rules have, however, resulted in a scalability problem. So, in an attempt to mitigate this problem, similar rules were replaced in [126] by general and less precise ones, and the resulting uncertainty was handled by probability, as explained in Section 2.2.1. This solution has improved the scalability but at the cost of producing many false positives, as reported by the authors.

Guéhéneuc *et al.* suggest that using a catalogue of DP recognition rules cannot, on its own, solve the problem of variants recognition, and proposed a constraint satisfaction based approach (i.e. DeMIMA) in which reverse-engineers can relax unsatisfied constraints to enable the recognition of variants not satisfying them [75]. This may suit the goal of their approach, which is to recognise design structures that are similar to DPs, but it is not an adequate approach if the intent is considered. Also, it may be argued that although the main purpose of DP recognition is to help reverse-engineers to understand the design of software systems, such an approach relies on their understanding of the design to decide which constraints to relax during the recognition process.

Defining recognition rules for the MARPLE's joiner module that "cover all possible variants" [173] is not a feasible task. In fact, the authors acknowledged in a previous article that some variants may need to be dealt with as if they were different DPs (i.e. having different recognition rules and filtering classifiers) [16].

Another mechanism implemented by MARPLE to tackle the variant problem is incremental learning, which enables users to add more instances to the training datasets and retrain the classifiers. It is assumed that, by changing the composition of the training datasets, the recognition can be focused on different variants [173]. The added instances can, however, introduce contradictory training examples, which will affect the quality of the datasets and, consequently, the performance of the classifiers.

In [161], the structural relationships (i.e. inheritance, interface implementation and aggregation) of potential DP instances are compared with those expected in order to calculate what is called *relation agreement value*, which is then combined with the output of the ANN used to reduce the search space to calculate *pattern agreement value*. Each potential instance is considered a true one if the latter value exceeds a threshold. Such a simple approach that relies on a limited number of structural relationships is highly likely to be ineffective in recognising and distinguishing variant instances from other random sets of similarly interconnected classes.

Although the trained classifiers used to filter out false positives (e.g. in Columbus [63]) and to reduce the search space (e.g. in [119]) can be capable of recognising DP variants (if properly designed and trained), this capability will be wasted if the core part of the recognition process is implemented by a method that cannot recognise them. The only existing approach in which the core recognition task is performed by trained classifiers is [39], which also enrich the class diagrams of software systems with information about indirect class relationships in order to enable the recognition of variants with indirectly connected classes. However, as discussed earlier, this approach has serious methodological problems that undermine its validity. The approaches proposed in [42] and [84] have made no attempt to recognise DP variants.

The difficulty encountered by existing recognition approaches when dealing with DP variants stems from the fact that they generally focus on the structural and behavioural characteristics of DPs which differ between different instances of the same DP. If more focus was, however, given to the intent of DPs, recognising variants would have become less of a problem since the intent does not differ from one instance to another.

#### 2.2.10.7 Limited Scalability:

Scalability is one of the main challenges to be addressed in DP recognition, and the success of any automated recognition approach depends highly on its scalability [123]. Many of the existing approaches suffer from scalability issues as the size of the analysed software systems increases, as noted in [13] [77].

Both [161] and the experiment reported in [77] use a search space reduction phase, and such an approach can be beneficial in improving the scalability. The only aim of the work proposed in [78], [140] and [119] is to provide a search space reduction phase that can be utilised by other DP recognition approaches. However, all these search space reduction phases suffer from other limitations as discussed above. In [39], the authors assumed that each DP instance must have at least one non-concrete class, and then used this inaccurate and restrictive assumption to reduce the search space. The size of the matrix representing the software system to be analysed is reduced in [164] by eliminating redundant rows and columns. In order to reduce the search space in [42], all classes that do not satisfy a minimum number of constraints are eliminated, which is likely to eliminate potential variant instances.

To overcome the scalability problem, similar recognition rules in FUJABA was replaced by general and less precise ones [126], as mentioned earlier. In MARPLE, the source code of software systems is not analysed directly but summarised first

into a set of predefined micro-structures [13]. The Columbus approach is, however, inefficient as it suffers from the combinatorial explosion problem [17]. The approach proposed in [84] is also inefficient as it uses expert systems, which normally employ an exhaustive search strategy [121].

#### 2.2.10.8 Subjective and Improper Evaluation:

One may have noticed that the discussion of the recognition approaches did not include a comparison of the recognition accuracy achieved by different approaches. The reason for this is that the majority of them are manually and subjectively evaluated, and this is generally the case for most DP recognition approaches in the literature [169]. Pettersson *et al.* have also found methodological problems in these evaluations which implies, as they state, "at least a limited generality of the conclusions drawn" [132, page 576], and they have also identified differences in several variables that make them incomparable.

One problem of evaluating recognition approaches subjectively is that they would naturally lead to overestimated accuracies. If the evaluators' understanding of the concept of a DP is represented as a circle, the boundary of which is specified by the recognition rules they have defined, the produced output is likely to fall within the boundaries of this circle, and hence the overestimated accuracies. Consequently, several recognition accuracies that have been reported by some authors have been disputed by others (e.g. the results reported in [160] have been disputed in [75]).

Although the peer-reviewed P-MARt repository [73], which is publicly available[15], has been created by repeatedly analysing the same set of software systems by multiple teams, it is still not widely used as an evaluation benchmark as it should have been. Among the approaches discussed in this chapter, the only ones that

[15]http://www.ptidej.net/tools/designpatterns/

have used it for evaluation is DeMIMA [75] and its related exploratory study [77] as well as the search space reduction approach in [140]. The only other approach that have apparently been evaluated objectively is FUJABA [125], which was evaluated based on the internal documentation of the test systems.

The approach in [39] uses the P-MARt repository as a training dataset as well as a test dataset which, besides its other methodological problems discussed in Section 2.2.5, invalidate any conclusions drawn. In the first phase of the Columbus approach [17], recognised instances were evaluated manually and subjectively by the authors. Its filtering phase [63], on the other hand, has been evaluated by performing cross validation on subjectively labelled datasets. The same applies to the MARPLE approach [173] as well as the approaches introduced in [161] and [119]. This method of evaluation means that what the reported results indicate is basically how well the classifiers have learnt the authors' subjective concepts of DPs, which may not necessarily be consistent with the objective concepts. Although the approach in [164] is a DP recognition approach, it was compared with a search space reduction approach, each of which was applied on different versions of the test systems. No test results were reported for the recognition approaches proposed in [42] and [84].

This lack of objective and proper evaluations and comparisons has made it difficult to know what approaches have performed better in what accuracy measure, which hinders real progress and improvement in the field. Moreover, the differences between different approaches in their recognition accuracy have never been analysed by using statistical tests. Although many approaches in the literature have claimed good, and sometimes almost perfect, recognition accuracy, such claims cannot be taken on their face value. In fact, several authors have recently acknowledged that the accurate recognition of DPs is still a problem that has not yet been effectively solved (e.g. [173], [137] and [25]).

Table 2.1 Summary of the discussed DP recognition approaches.

| DP Recognition Approach | Intent Reasoning | Design Information | Subjectivity In Recognition Rules | Training Dataset | Variant Recognition | Scalability | Evaluation (Source of Test Set) |
|---|---|---|---|---|---|---|---|
| **FUJABA** [125] [126] | Not considered.* | Limited set of binary (structural and behavioural) features. | Subjective rules over subjectively selected features. | N/A• | A rule is needed for each variant to be recognised. | Relies on solving subgraph isomorphism problem which is NP-complete. | The internal documentations of the test systems. |
| **Columbus** [17] [63] | Limited support in filtering phase. | Limited set of intra-class metrics & binary (structural and behavioural) features. | Subjective rules over subjectively selected features. | Instances passed by the first phase are labelled subjectively & used to train filters. | A rule is needed for each variant to be recognised. | Suffers from Combinatorial explosion problem. | Subjective evaluation of recognised instances. |
| **MARPLE** [173] [172] | Explicitly ignored during dataset labelling. | About 70 binary micro-structures used as features. | Subjective rules over subjectively selected features. | Instances passed by the first phase are labelled subjectively & used to train filters. | Some variants need to be treated as if they were different DPs. | Uses subgraph isomorphism but it is claimed to have linear complexity. | Cross validation on subjectively labelled datasets. |
| **PTIDEJ** [75] [78] [77] | Limited support in search space reduction phase. | Limited set of intra-class metrics & only binary structural features. | Subjective rules over subjectively selected features. | Search space reduction classifiers are trained by a small repository of DPs (P-MARt). | Recognition constraints can be relaxed manually. | Using space reduction phase helped to circumvent the combinatorial explosion. | Objective evaluation based on the P-MARt repository. |
| **SVM** [39] | Not considered*, but can have a limited intent reasoning support in filtering phase. | 45 intra-class metrics with a complete disregard of the type & direction of class inter-relationships. | Subjectively selected features. | Recognition classifiers are trained by the P-MARt but with corrupting permutations. | Enriches the class diagram with information about indirect class relationships. | Made & used an incorrect assumption to reduce the search space. | Evaluated on exactly the same dataset used in training (P-MARt). |
| **ANN** [161] | Not considered*, but can have a limited intent reasoning support in filtering phase. | Limited set of 12 intra-class metrics and 4 binary structural features. | Subjective rules over subjectively selected features. | Search space reduction classifiers are trained by manually constructed dataset. | Uses a measure called pattern agreement value, over which a threshold is set. | Uses a search space reduction phase. No discussion of its effect is reported. | Cross validation on subjectively labelled datasets. |
| **Fuzzy Matching** [164] | Assumed incorrectly that the sequence diagram provides enough information to discern the intent. | Limited set of binary structural features as well as behavioural features (sequence diagram). | Subjective rules over subjectively selected features. | N/A• | Attempts to recognise variants by devising a simplistic & ill-designed inference method. | The size of system matrix is reduced by eliminating redundant rows and columns. | Compared with other approaches based on inconsistent versions of test systems. |
| **Hierarchical Clustering** [42] | Not considered*. | Limited set of binary (structural & behavioural) features. The direction of relationships is disregarded. | Subjective rules over subjectively selected features. | N/A• | Not considered*. | Eliminates software classes that do not satisfy a minimum set of constraints. | No test results are reported. |

*Not (intentionally) addressed nor discussed. • Not Applicable.

#### 2.2.10.9 Evaluation Summary:

The main DP recognition approaches discussed above are summarised in Table 2.1. In cases in which the approach involves a pre- or post-processing step, the information presented in the table focuses more on the core step of the recognition process.

## 2.3 Studies on the Quality Impact of DPs

The design principles on which the DPs are based, as well as the quality improvements expected from their application, were discussed in Section 2.1. These quality improvements are, however, only theoretically presumed and they may not necessarily reflect the real and practical impact of implemented DPs. So, such presumptions should not be relied upon until they are supported by a solid empirical evidence [134]. Only after such a solid evidence is available, we may develop a theory based on which predictions can be made as to what software quality aspects are likely to improve/deteriorate with different DPs. So far, however, there is not much work in the literature that empirically validate their impact [87] [175].

This section discusses some of the attempts reported in the literature to evaluate the quality impact of DPs and, then, discuses some of their main limitations. However, before that, models and metrics for software quality evaluation are discussed first.

### 2.3.1 Quality Models and Metrics

There exist several software quality models (e.g. ISO/IEC 9126 ) which can be used to evaluate the quality of software systems as well as the quality impact of DPs. These models are usually defined hierarchically in terms of characteristics (e.g. maintainability) and sub-characteristics (e.g. correctability and expandability).

These characteristics can be classified as external quality attributes. They are *external* in the sense that they cannot be evaluated based only on the software artefacts but also based on how the artefacts interact with their environment [120]. For example, the fault-proneness external attribute can be evaluated by using the number of post-release faults identified.

There are, on the other hand, internal quality attributes (e.g. coupling, cohesion and complexity) [120]. Each of these attributes can easily be measured by metrics that are calculated based only on software artefacts. However, such metrics are of little or no value unless they have a validated association with an external attribute of interest [59]. For a metric to be validated, it has to go through two levels of validation [60]. The first is the internal validation in which it should be shown, through a theoretical exercise, that the metric properly captures the internal attribute it claims to characterise. The second level is the external validation in which the metric is shown empirically to be associated with an external attribute.

Since the internal metrics can be calculated easier and earlier than external metrics, they can be used as early predictors of external quality attributes, and hence they have attracted a large interest from researchers in the field of software engineering. Many internal OO metric suites and models (e.g. Chidamber and Kemerer [38] and QMOOD [18]) have been proposed and validated against various external attributes like maintainability [45], change-proneness [114] and fault-proneness [136].

### 2.3.2 DP Impact on Software Quality

The external and internal metrics discussed in the previous subsection have been used in empirical studies to evaluate the quality impact of DPs. All of these empirical studies fall into two of the three broad categories of experimental methods in the taxonomy proposed by Zelkowitz and Wallace [174]. The first is the controlled

experimental method category, to which the majority of these studies belongs [7]. The outcome measures in these experiments can be an external metric (e.g. number of failing tests) or an internal metric (e.g. the cyclomatic complexity) of the piece of software produced [138].

There are, however, problems associated with the use of controlled experiments to evaluate the quality impact of DPs. Using student subjects, which most of them do [138] [175], reduces the practical value of these experiments [101]. Also, the results of these experiments, which are normally conducted in a small artificial settings by using simple materials, may not transfer to complex (e.g. industrial) settings [174] [101]. Moreover, the fact that many, if not most, of these experiments are using standard DP instances means that their results may at least not be generalised to DP variants [7].

The historical method category is the second category to which the rest of the empirical studies belong. They are *historical* because they are based on analysing data collected from existing software systems, and they can be sub-categorised into *Legacy Data* and *Static Analysis* based methods, according to the Zelkowitz and Wallace's taxonomy. *Static Analysis* methods are based only on software source artefacts, which are their sole source of information, and in such a case, only internal metrics can be calculated. *Legacy Data* based methods, on the other hand, also use other sources of data such as version control systems and bug tracking databases, which makes it possible to calculate external metrics as well.

The discussion in this section will be limited to studies in the historical-method category. This is because they are the ones that are closely related to the work in this thesis, which will also evaluate the quality impact of DPs by following the static analysis based historical-method approach. The following discusses existing studies in three subsections, each of which discusses studies that analyse the impact of DPs on a certain external quality attribute.

#### 2.3.2.1 Maintainability

In a study that involves only one software system, Hegedus *et al.* analysed the impact of applying DPs on software maintainability by using 258 revisions of the system [87]. The exact impact of interest here is that of all applied DPs together at the system as a whole. Information about implemented DP instances were collected from the documentation, relying on the assumption that every instance is explicitly documented. A probabilistic quality model was used to estimate and quantify the maintainability based on some internal metrics. The number of applied DPs was found to change in only 5 revisions, four of which the number increased and so was the estimated maintainability score. The authors also studied the relationship between pattern line density (i.e. ratio of the number of code lines that are in DP instances) and the estimated maintainability scores for all of the 258 revisions, and they found that they were strongly correlated. However, the authors stated that the model used to estimate the maintainability is not fully validated.

The impact of DPs on the maintainability of game applications was studied by Ampatzoglou and Chatzigeorgiou [8]. The study was based on multiple versions of two open source game programs implemented in two different programming languages. DP instances were recognised manually in one program and by using a tool in the other, and their impact was evaluated based on OO metrics. The authors reported a positive change on the values of these metrics. However, the reported results are based on the metric values of only two classes, one from each program, as well as the average of metric values at the system level. Besides the fact that two individual classes are too few to draw a conclusion based upon, using the average may lead to misleading conclusions given that most of the OO metrics have a skewed distribution [113] [93].

#### 2.3.2.2 Change-Proneness

Ampatzoglou *et al.* [9] studied the stability of DP participant classes. The study was based on a dataset of DP instances from 537 open source software systems, which were obtained by merging the results of two DP recognition tools. The stability of classes was quantified by a measure called Ripple Effects Measurement (REM). Five different statistical analyses were performed on different subsets and combinations of the dataset. The authors claim that the results indicate that DPs do have an effect on the stability of software classes. However, the conclusions drawn are not always supported by the statistical analyses used. In fact, in one case for example, despite finding that a pair of variables were "almost not correlated at all", the authors went on and suggested that they were correlated based on a line chart. Also, although some statistically significant relationships were found, they can be produced by mere chance when multiple statistical tests are performed on different subsets and combinations of the same dataset [101], which may be the case here.

Software systems undergo continuous changes as they evolve, which can lead their internal structure to decay. The extent to which this decay affect DPs is investigated in an exploratory study by Izurieta and Bieman [92]. Two types of decay have been identified for DPs: pattern rot, in which system changes violate the intent and the key concept of the instantiated DP, and pattern grim, in which system changes add irrelevant artefacts to DP instances. The pattern rot was measured by inspecting instances manually for violations, while the pattern grim was measured semi-automatically by using coupling and some other metrics. Multiple versions of two development and one database systems were included in the study. DP instances, which were automatically recognised in these systems, were checked manually and only instances that clearly demonstrate an intentional implementation of DPs were kept. The study found evidence for pattern grim, which was attributed to coupling increase, but no evidence was found for pattern rot. However, this

conclusion is not supported by a statistical test due to the small size of the dataset (only few instances were found for 7 DPs).

To test whether DP instances support, in practice, the open-closed principle as it is theoretically assumed, Bieman *et al.* studied three proprietary and two open source systems [23]. The documentations were searched manually for DP instances, and the implementation details of each found instance was examined to check if it was a true instance. The number of changes made to all classes during a period of time was used as a *change* measure. The Mann-Whitney statistical test was then used to compare the number of changes made to DP participant classes and non participant classes. The results showed that participant classes were more change prone in four out of the five systems. The results held up even after accounting for the effect class size as a potential confounding factor.

Di Penta *et al.* conducted a study on the change proneness of DPs but focused on the differences between individual roles within DPs rather than the differences between DP participant and non participant classes [49]. The authors also studied the types of changes and whether there are certain changes associated with certain roles. The study was performed on three open source systems, each of which represents a different domain. A DP recognition tool was used to recognise instances for 12 DPs in these systems. The change information was extracted from hundreds of snapshots for each system and their history logs. The Kruskal-Wallis test was then used to compare the number of changes made to classes playing different roles in each DP. The results were said to be, in most cases, in accordance with the intuitive expectations.

Khomh *et al.* conducted a study in which they analysed the effect of the number of roles played by classes on some of their internal OO metrics as well as their change proneness [99]. The roles considered represent six DPs, instances of which were recognised by using a tool followed by a peer-review inspection by the authors.

The OO metrics calculated for the classes include coupling, cohesion and complexity metrics. Their change prone was measured by counting the number and frequency of changes applied to them. Out of the total 56 metrics, 29 and 48 metrics were found to be statically different (based on the Wilcoxon rank-sum test) in classes playing one and two roles, respectively, compared to classes playing no roles. These differences were mainly not in the desirable direction. It was also found that classes playing one or two roles are more change prone than classes playing none.

#### 2.3.2.3 Fault-Proneness

The impact of applying DPs on the fault-proneness of DP participant classes was studied by Vokac [163]. The study was based on analysing the evolution history of a large commercial software system. The history includes three years of weekly versions of the system as well as the bug tracking database in which all reported bugs are recorded. Instances for five DPs were recognised by using a tool developed specifically for the system under study, and hence, some recognition rules were tuned to its coding style. All recognised instances were manually examined to remove false positives. Also, non recognised classes were randomly sampled, by using a statistical sampling method, to estimate the false negative rate, which was found to vary between 2% and 10% for different DPs. The logistic regression model was then used to estimate the association between the DPs and the occurrence of at least one fault. The results showed different, positive and negative, associations for different DPs.

To answer the question of whether or not DP participant classes are more fault-prone than other non-participant classes, Gatrell and Counsell studied a commercial proprietary software system [72]. Instances for 13 DPs were searched for manually in the documentation, and each found instance was verified by inspecting its implementation. Fault information was collected from the source control system.

The average number of faults for DP participant classes was found to be higher than the average of non participant classes. It was also found that the average of faults differ for different DPs. However, no statistical tests was performed to check if these differences are statistically significant.

### 2.3.3 Discussion

The main shortcomings and limitations identified in the studies discussed in this section are summarised in the following subsections.

#### 2.3.3.1 Inaccurate Datasets:

In order to study the impact of a set of DPs on a software system, all of their existing instances need to be recognised first. Two recognition approaches have been used for this purpose. The first is using DP recognition tools, which is the approach adopted by [8], [9] and [49]. The main problem with using recognition tools is the accuracy of their results. In [9], for example, the results of two tools were merged although they admittedly have, on average, less than 50% recall and precision. This is a serious threat to the validity of any conclusion drawn in these studies, and that is why, in the study reported in [92], the automatically recognised instances have been manually inspected to remove any false positives in order to minimise this threat. The same applies to the study reported in [163] but, since its analysis includes non DP participant classes, any unrecognised classes (i.e. false negatives) would be analysed as if they were not DP participants. The best study in avoiding this shortcoming is [99] in which the analysis was based only on a peer-reviewed sample of the tool's output.

The second approach by which DP instances are recognised is through searching the source code and design documentations. This was done manually in [8], [23] and [72], and automatically by using a text parser in [87]. The problem with

this approach is that instances which are not (properly) documented may not be recognised. In fact, Ferenc *et al.* assume that the design of any high-quality software system is likely to have occurrences of DPs even if they are not intentionally introduced [63], and any such instances will obviously not be documented. This problem gets worse when the documentation of a large system is searched manually as it is the case in [23].

#### 2.3.3.2 Measuring Impact on System Level:

In [87] and [8], the impact of applying DPs is measured at the system level. Given that there are many and complex factors that could have an effect on the system, any impact captured at the system level may be difficult to trace and attribute to the application of some DPs.

#### 2.3.3.3 Mixing Multiple DPs and DP Roles:

Comparing participant classes of multiple DPs together versus non participant classes, as it is the case in [9], [23], [99] and [72], is a problem for the following three reasons:

- Since participant classes play different roles in different DPs, they are expected to be characteristically different with different expected quality improvements. Client roles, for example, can be played by any class in the system, and hence, they are not expected to exhibit any particular quality characteristics. The Subject role in the Observer DP, on the other hand, can only be played by classes that are designed to fulfil a certain task in a way that makes them less coupled to individual observing classes. So, combining them despite of their differences may obscure their individual characteristics.

- There are also problems in the conclusions drawn when such a participant/non-participant comparison is made. Since the set of participant classes will natu-

rally be composed of different number of classes that happen to play different roles, conclusions drawn may actually reflect their proportions more than they reflect the quality impact of the DPs.

- Attempting to draw a general conclusion in this way ignore the possibility of having participant classes of unconsidered DPs, which would obviously be in the non-participant set.

The first two reasons also apply to when the roles of a single DP are combined in order to analyse its impact as a whole, which was done in [9], [163], [92] and [72]. That is because even roles within an individual DP may still be characteristically different due to differences in the responsibilities assigned to each one.

#### 2.3.3.4 Not Using a Baseline:

When the comparison is not made with non participant classes as a baseline, no information can be derived about how DP instances are different from other parts of the system. This is the case in [92] where all participant classes of different DPs are treated as a single group. This is also the case in [9], [49] and [72] where the participant classes of each DP are compared to each other (i.e. a role versus another role), or different DPs are compared based on all of their role-playing classes.

#### 2.3.3.5 Small Sample Size:

All the studies discussed are based on only six or less software systems (mostly three or less). The only exception is [9] that it is based on 537 systems which, however, suffers from several shortcomings and limitations with regards to the accuracy of the dataset extracted from these system, as discussed above, as well as those noted in Section 2.3.2.2. In the only study that uses six systems [99], only 238 classes have been sampled and included in the analysis.

Relying on a small number of systems limits the generalizability of any finding. The generalizability in [23] is also limited by the fact that three of the five systems included in the analysis are developed in the same company, which probably makes this study more relevant to the development experience, practices and methodology in this particular company.

#### 2.3.3.6 Limited/Improper Use of OO Metrics:

Despite the wealth of available OO metrics, there have been a surprisingly limited number of studies on the impact of DPs on their values. This can be observed in the studies covered in this thesis as well as in a recent mapping study in which it was suggested that "assessing the effect of patterns through code and design metrics is a field that needs further investigation and is expected to grow in the next years" [7, page 1957].

In one study [87] in which metrics have been used, their values have been measured at system level for multiple revisions of a software system and correlated with maintainability scores estimated by a not fully validated model. In [8], several OO metrics were also calculated at the system level by using their average values, which is not a proper way of summarizing skewed data, as mentioned earlier. In [92], metric values have been measured at realization (i.e. instance) level and the conclusions drawn were not supported by a statistical test. Although metrics were properly calculated for individual classes in [99] and their values were compared using a statistical test, classes playing different DP roles were merged into a single group, which has the drawbacks explained above. Also, this latter study, as well as the formerly referenced study [8], uses different versions of the LCOM (i.e. Lack of COhesion in Methods [38]) metric to measure class cohesion which, however, suffers from several problems and anomalies, as will be discussed in Chapter 4 (Section 4.1.2).

In [9], five OO metrics were used to calculate a single stability measure on which statistical tests were performed. The analysis of the impact of DPs in the other studies (i.e. [163], [72], [49] and [23]) was not performed on source code and design metrics, but rather on external quality attributes (e.g. fault-proneness as measured by the number of faults).

## 2.4 Conclusions

This chapter has discussed several approaches to DP recognition as well as to evaluate their impact on software quality. Most of these approaches suffer from several shortcomings and, sometimes, even methodological problems. While the quality of empirical software engineering research has been described by Kitchenham *et al.* [101] as being of poor standards, research in these two sub-fields generally seems not to be an exception. Each of the following chapters will consider and contribute to solving one or more of the limitations identified in this chapter.

# Chapter 3

# Structural and Behavioural Features

In any pattern recognition system, finding the subset of features in which "the true (unknown) model of the patterns can be expressed" is critical and crucial for its success [57, page 7], and DP recognition is not an exception. There is, however, an acknowledged lack of research on such subsets for DPs. In fact, there has been no attempt to even define a global set of features from which different subsets can be objectively selected for different DPs. Consequently, the authors of different previous recognition approaches have defined their own recognition rules over the design aspects that they perceive as relevant for the DP at hand [52]. The main goal and contribution of this chapter is to fill this gap by introducing a new pool of features, or a global set of features, that capture various structural and behavioural aspects of OO software design. The term *feature* set, as in pattern classification terminology, refers to the set of descriptors which are, in this case, used to represent design information. However, since many of these features are basically OO metrics, the terms *metric* and *feature* are used interchangeably.

The chapter starts by setting the criteria for the feature set to be defined. Then, the following two sections introduce and discuss features that, respectively, capture structural and behavioural aspects of OO software design. Then, in Section 3.4, the

importance of taking the context into account for inferring the intent of potential DP instances is discussed and an alternative feature calculation method is proposed for this purpose. Section 3.5 discusses the outcomes of this chapter and how the work presented here relates to the rest of this thesis.

## 3.1 Criteria For the Feature Set

The global set of features should be developed with two main criteria in mind. First, its features should not be designed around particular DPs but, instead, around the aspects and properties of software design in order to provide as rich design information as possible. This criterion helps to avoid being limited by what is assumed, based on DP descriptions and personal interpretations, to be relevant for a certain set of DPs. It is also particularly helpful for inferring the intent as it may not be known beforehand which features would hold traces of relevant intent information. Some features may, however, happen to be relevant to one or more DPs in a theoretically intuitive way, but others may not have an obvious or immediate relevance to any DP. In order to meet this criterion, a set of aspects and sub-aspects of software design has to be identified. A set of features are then defined for each identified aspect to capture all of its properties, and these features are grouped into different *degree* types. *Degree* as a term is used in this thesis to emphasis the importance of capturing design properties in *degrees* (i.e. not in a binary form), which provides more design information.

The second criterion is that the included features should provide as accurate representation of the design of software systems as possible. In order to meet this criterion, the features have to be defined at a fine-grained level and be unambiguously recoverable. The features need also to be formally defined as even simple metrics, like the number of methods, can be ambiguous [149]. This ambiguity has

caused inconsistency problems between different software engineering datasets [139]. The formalism used to define the metrics, which is based on set theory, is programming language independent. This formalism is provided in Appendix A and can be referenced as required. However, some definitions are unavoidably language dependent due to differences in the concepts and constructs available in each language. In such cases, features are defined for Java, which is the language of all systems in the training and testing datasets, but can easily be adapted for other languages.

Although the feature set is not designed around particular DPs, examples of how could these features support the recognition of some DPs are given to illustrate the relevance and importance of the property captured. While most examples given represent variants of the Adapter DP in order to give a flavour of the variant problem by showing various ways in which a single DP can be implemented, examples from other DPs are also used where appropriate.

## 3.2 Structural Features

Features in this section capture the structural aspects of software design at intra and inter class levels. While features of the former are mainly OO metrics calculated for individual classes, features of the latter represent different structural relationships between them.

Although most of the OO metrics in this section are not new and are not claimed to be so, references are not provided for each single one as they would be too numerous. The interested reader can find references to the original definitions in [136] and [168]. Their formal definitions in this thesis may, however, differ from the original sources due mainly to clarifying ambiguities in their definitions. The main contribution of this section is in organizing them into different degree types

capturing different design aspects, and using them as building blocks to construct a part of the global feature set.

### 3.2.1 Intra-Class Features

In this subsection, the features used to represent software design at the class level are introduced and discussed. These features are mainly metrics calculated for individual classes to represent different properties of their internal structure. The metrics are grouped together into different *degree* types as follows:

#### 3.2.1.1 Degree of Accessibility

Access to the attributes and methods of OO classes can be determined by using access control modifiers. The access control modifiers set the access level permitted to the attributes/methods of a class, which include *public*, *protected* and *private* modifiers. The number of methods with different modifiers gives an indication for the intended purpose of a class. Adapters, for example, are expected to have public methods but probably no protected ones in order to make the adapted methods accessible by client classes. The number of public methods ($PubM$) in a class $c \in C$, where $C$ is the set of all classes in a software system, is defined as follows:

$$PubM(c) = |\{m \mid m \in M_{DEC}(c) \ \wedge \ Public(m)\}| \tag{3.1}$$

Metrics for methods with other modifiers can be defined following the same template, they only require substituting the $Public()$ predicate in Eq. (3.1) with the appropriate predicate (i.e. $Private()$ or $Protected()$). The same applies for the corresponding attribute metrics which also require substituting $M_{DEC}(c)$ with $A_{DEC}(c)$.

#### 3.2.1.2 Degree of Virtuality

Methods that are declared in a class but without any implementation are called *abstract* or *pure virtual* methods. Classes with one or more *abstract* methods may not be playing the Adapter role whereas such classes may play the Target role, and in such a case the concrete (i.e. non-abstract) methods may be used to provide default implementations. Two *Degree of Virtuality* metrics are defined. The first metric captures the number of methods that are *abstract*. The definition of this metric follow the same template as the $PubM$ metric above, it only requires substituting the $Public()$ predicate in Eq. (3.1) with the $Abstract()$ predicate. The second metric is a binary metric that captures whether or not the class itself is an *abstract* class. Classes are considered *abstract* if they are declared as such or if they have one or more *abstract* methods that makes them non-instantiatable.

#### 3.2.1.3 Degree of Availability

The *static* modifier can be used to determine whether an attribute/method is available at the class level, and hence can be accessed by all of the instantiated objects of a class, or at individual objects level. While it is possible to implement the Adapter DP with static attributes and methods, the same may not apply to the Composite DP, the role of which is to compose different sets of objects in each instantiated object. Two *Degree of Availability* metrics will be used, one for methods and the other for attributes. The definition of these two metrics follow the same template as the definition of $PubM$ metric in Eq. (3.1) above, with the substitution of the relevant predicates.

#### 3.2.1.4 Degree of Contribution

Metrics in this category measure class contribution mainly in terms of methods and method implementations. There are different possible forms for this type of

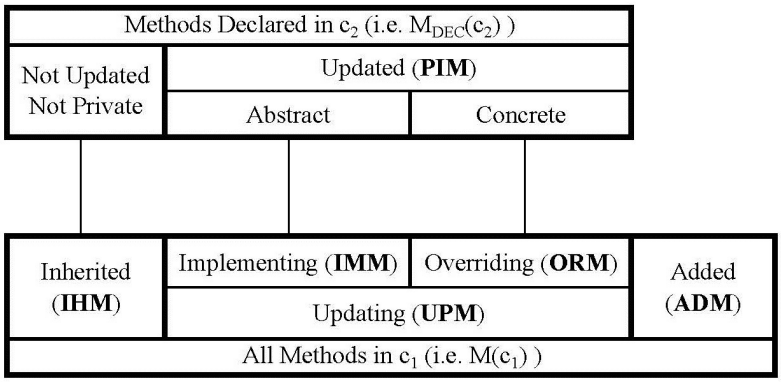



Fig. 3.1 Different forms of method contributions by class $c_1$ and its superclass $c_2$.

contribution, which can be seen in Fig. 3.1 in which the acronyms between brackets refer to the metrics defined to capture the corresponding contribution[1]. The set of metrics defined to capture these forms of method contributions are as follows:

- **Number of Overriding Methods (*ORM*):** This metric measures how many of the methods declared in a class overrides inherited non-abstract methods. The methods in an Adapter class can, for example, override Target class methods that may provide default implementations. Method overriding mechanism can also be used to wrap/decorate super-class methods with additional responsibilities, as it is the case in the Decorator DP. The number of overriding methods in a class $c \in C$ can simply be defined as follows:

$$ORM(c) = |\{m \mid m \in M_{OVR}(c)\}| \tag{3.2}$$

- **Number of Implementing Methods (*IMM*):** This metric measures how many of the methods declared in a class implement an abstract method in a super class. Since one of the design principles implemented by DPs is *program to an interface not an implementation* principle, instances of almost all DPs are likely

[1]This figure is adopted and adapted from Fig. 2 in [157].

to have such implemented methods in at least one of their participant classes. A conventional implementation of the Adapter class, for example, would only have methods that implement abstract Target class methods. The definition of the $IMM$ metric is similar to that in Eq. (3.2) but with the $M_{OVR}(c)$ set replaced by $M_{IMM}(c)$ set.

- **Number of Updating Methods ($UPM$):** The two metrics above (i.e. $ORM$ and $IMM$) capture similar forms of contribution as they both measure the number of methods declared in a class that provide an update for a correspondent method declared in a super class. Given the similarity of the properties captured by them, the UPM metric is added to combine them into one metric. This combined metric may actually be useful in cases where the Adapter class, for example, have methods that override default Target methods and, simultaneously, other methods that implement abstract Target methods. The $UPM$ metric can simply be calculated by summing the previous two metrics.

- **Number of Added Methods ($ADM$):** This metric measures how many of the class methods are totally *new*. In other words, the number of declared methods that do not override or implement methods already declared by a super class. Since this metric captures the contribution of a class to its method interface, it does not count private methods. Typical Adapter classes, for example, may not add new methods but only implement/override already declared ones. Replacing $M_{OVR}(c)$ set in Eq. (3.2) with $M_{NEW}(c)$ produces the definition of $ADM$ metric.

- **Number of Polymorphically-Invocable Methods ($PIM$):** This metric measures how many methods in a class have been *updated* by a subclass. The implementation of such *updated* methods can be polymorphically substituted by the implementation of their *updating* counterparts. The Number of Poly-

morphic Methods (NOP) metric in [18] is similar to this metric but it is limited to virtual/abstract methods. In the Decorator DP, for example, all methods in Component classes should ideally be polymorphically-invocable methods. This metric can be defined for classes $c_1, c_2 \in C$ as follows:

$$PIM(c_1) = |\{m_1 \mid m_1 \in M_{DEC}(c_1) \wedge \neg Private(m_1) \wedge \exists m_2 \in M_{DEC}(c_2) : Generalize(c_2, c_1) \wedge SameSignature(m_1, m_2)\}| \tag{3.3}$$

where the predicate $SameSignature(m1, m2)$ is true iff $m_1$ and $m_2$ methods have the same name and the same set of parameters in the same order. The predicate $Generalize(c_2, c_1)$ is defined in Section 3.2.2 below.

- **Number of Final Methods/Attributes ($FIM$/$FIA$):** These metrics measure how many methods/attributes are declared as *final*. When a method is declared as *final*, its implementation cannot be overridden by subclasses. Although it is technically possible for a method, at least in Java, to be declared as *final* as well as *private*, it does not make sense because *private* methods cannot, by definition, be overridden or even accessed by sub-classes, and so, *private* methods are not counted with the *final* ones even if they are declared as such. The $FIM$ metric is defined as follows:

$$FIM(c) = |\{m \mid m \in M_{DEC}(c) \wedge Final(m) \wedge \neg Private(m)\}| \tag{3.4}$$

For example, a class in which all methods are declared as *final* cannot be a Target or a Component class. When an attribute is declared as *final*, the value initially assigned to it cannot be changed later. This metric attribute (i.e. $FIA$) is an example for features that do not have obvious or immediate relevance to

any DP, and it can simply be defined by substituting $M_{DEC}(c)$ and $Public()$ in Eq. (3.1) with $A_{DEC}(c)$ and $Final()$, respectively.

- **Number of Inherited Methods/Attributes ($IHM$/$IHA$):** These metrics measure how many methods/attributes a class inherits and, in the method case, does not override or implement. The value of the IHM metric in a typical Adapter class, for example, can be zero as its basic role is to implement Target class methods. The definition of the $IHM$ and $IHA$ metrics can be produced by respectively substituting the $M_{OVR}(c)$ set in Eq. (3.2) with $M_{INH}(c)$ and $A_{INH}(c)$ sets.

#### 3.2.1.5 Degree of Ancestry

The position of classes in the inheritance tree gives an important information about their role within the inheritance tree where they reside. Classes at the top level are more likely to represent general types that are implemented, specialized or extended by those at lower levels. Two *Degrees of Ancestry* metrics will be used to capture this property: Depth of Inheritance ($DIT$) [38] and Class-to-Leaf Depth ($CLD$) [156], which measure the longest path to the root or leaf classes, respectively. In the Adapter DP, for example, while Target classes may have a value of zero for the $DIT$ metric, Adapter classes are expected to have the same value but for the $CLD$ metric.

### 3.2.2 Inter-Class Features

Features in this subsection capture the structural relationships between classes. There are two main types of such relationships: *generalization* and *association* relationships, which are discussed in the following two subsections.

#### 3.2.2.1 Generalization Relationships

Generalization is the taxonomic relationship between classes [141]. These relationships can be used as a mechanism for reuse where subclasses build on and extend their super classes. They can also be used to achieve type matching where objects of different classes can dynamically substitute one another as long as the classes belong to the same family (i.e. share a super class), which is known as *polymorphism*. Generalization is a common and conventional OO concept that are used in most DPs.

Although the definition of all the previous features are programming language independent, it is unavoidable here to be less-language independent in order to properly and accurately capture the different possible types of generalization relationships in Java. While Java explicitly distinguish *interfaces* from other regular classes with a specific keyword, other languages (e.g. C++) do not distinguish them. So, only in this subsection, the terms *class* and *interface* have different meanings.

There are two mechanisms in Java to realise the generalization relationship: class extension and interface implementation. Four binary and oriented features will be used to capture generalization relationships instantiated by using these two mechanisms, and they are defined by the following predicates for classes/interfaces $c_1, c_2, c_3 \in C$:

- $Super(c_1, c_2)$ which is true iff the class/interface $c_1$ directly extends the class/interface $c_2$, respectively.
- $Ancestor(c_1, c_2)$ which is true iff the class/interface $c_1$ directly or indirectly extends the class/interface $c_2$, respectively.
- $Interface(c_1, c_2)$ which is true iff the class $c_1$ directly or indirectly implements the interface $c_2$. Interfaces can be implemented indirectly when a directly implemented interface ($c_3$) extends another interface ($c_2$).

- *Generalize*$(c_1, c_2)$ which is true iff the class/interface $c_1$ has any direct or indirect generalization relationship, through class extensions or interface implementations, with the class/interface $c_2$.

The Adapter DP, for example, can be implemented with a Target interface that is implemented by an Adapter class. It can also be implemented with a Target class, in which some methods are declared as *abstract* while default implementations are provided for others. The Adapter class, in the latter case, implements the abstract methods and (possibly) override the default ones. Any large dataset for the Adapter DP, or any other DP with a generalization relationship, should include instances of both cases. The decision with regards to which one of the four generalization relationships is more relevant than the other for a particular DP, and their respective weights in the classification function that classifies potential DP instances, will be objectively determined based on the dataset of each DP.

#### 3.2.2.2 Association Relationships

The structure of OO systems is glued together by association relationships [141]. In its general form, the *association* relationship is defined as the mechanism that enables communication between objects of different (and also the same) classes. In cases when an association relationship represents a part-whole relationship, it is called *aggregation*. In the special case of *aggregation* when the part has a lifetime dependence on its whole, and it is exclusively owned and accessible by its whole, the relationship is then called *composition* [129].

The three types of association relationships above are the ones often used in existing DP recognition approaches. However, the difference between these types can sometimes be ambiguous [129]. The *aggregation* and *association* relationships, in particular, are often distinguished based on taste rather than semantic differences [141], and some go even further by suggesting that *aggregation* is strictly meaningless

[68]. The ambiguity becomes even more problematic at the implementation level where there is no constructs by which the specified relationships can be realized, and both the *associations* and *aggregations* relationships are implemented as fields referencing the associated/aggregated classes [142], which hinders the ability to distinguish and recover them accurately from implementations. Although that *composition* relationships, on the other hand, may relatively be more distinguishable, capturing and verifying the existence of such relationships is a challenging task that may require dynamic analysis [74].

Although the problem of distinguishing between these types of association relationships is acknowledged in the GoF book [71, pages 22-23], it has not yet been dealt with and properly addressed. Instead, these types of associations are sometimes given different definitions (e.g. [11][17][125][133]), not defined at all (e.g. [164]) or approximated and combined into one type (e.g. [115]).

To overcome this problem, a new set of relationship features is proposed that covers different types of association relationships. The new set includes features that are unambiguously defined and easily recoverable, and they also provide more information about the association relationships between classes than the aforementioned three ones. These features are binary, oriented and intransitive, and they are defined as follows:

- **Association Through Method Parameter (*ATMP*)**: this relationship is defined by the predicate $ATMP(c_1, c_2)$ which is true iff $\exists p \in PAR(M_{DEC}(c_1)) \wedge Type(p, c_2)$, where $PAR(M_{DEC}(c_1))$ is the set of method input-parameters in the set of methods $M_{DEC}(c_1)$, and $Type(p, c_2)$ is the predicate which is true iff the parameter $p$ has the static class type of $c_2$.

- **Association Through Method Return-Type (*ATMR*)**: this relationship is defined by the predicate $ATMR(c_1, c_2)$ which is true iff $\exists r \in RET(M_{DEC}(c_1)) \wedge$

$Type(r, c_2)$, where $RET(M_{DEC}(c_1))$ is the set of method return-parameters in the set of methods $M_{DEC}(c_1)$.

- **Association Through Constructor Parameter (*ATCP*)**: this relationship is defined by the predicate $ATCP(c_1, c_2)$ which is true iff $\exists p \in PAR(CON(c_1)) \wedge Type(p, c_2)$, where $PAR(CON(c_1))$ is the set of method input-parameters in the set of constructor methods $CON(c_1)$.

- **Association Through Member Attribute (*ATMA*)**: this relationship is defined by the predicate $ATMA(c_1, c_2)$ which is true iff $\exists a : a \in A_{DEC}(c_1) \wedge \neg Array(a) \wedge Type(a, c_2)$.

- **Association Through Array Attribute (*ATAA*)**: this relationship is defined by the predicate $ATAA(c_1, c_2)$ which is true iff $\exists a : a \in A_{DEC}(c_1) \wedge Array(a) \wedge Type(a, c_2)$.

- **Association Through Object Instantiation (*ATOI*)**: this relationship is defined by the predicate $ATOI(c_1, c_2)$ which is true iff an instance of the class $c_2$ is created in the class $c_1$.

Different subsets of these relationships are expected to be realised in the instances of different DPs as well as variants of the same DP. Adapter classes may, for example, have member attributes of Adaptee class types, which are initialized by Adaptee objects received as constructor parameters. Alternatively, Adaptee classes may be instantiated inside individual Adapter class methods, and the objects created are then used to call the methods to be adapted.

## 3.3 Behavioural Features

Features in this section capture the behavioural aspects of software design in terms of method invocations between classes. There is not, however, many types of

method invocations that can be distinguished, and consequently, there is not much information that can be recovered about them. To solve this problem, another factor has to be considered in order to distinguish what would otherwise be the same, which generates more information about the nature and the purpose of method invocations between classes. One, and probably the only, relevant factor to be considered is the relationship between the enclosing classes of the invoking and the invoked methods. If, for example, a class invokes a method in a sibling class, such an invocation may signify a different intent or purpose than if the method invocation was between a class and a totally unrelated class.

Different types of invoker/invokee relationships are discussed in Section 3.3.1. Then, Section 3.3.2 discusses the types of the method invocations themselves. These invoker/invokee relationships as well as the method invocation types can be seen as the dimensions on which method invocations differ, which are then used in Section 3.3.3 to produce the behavioural features to be included in the global feature set.

### 3.3.1 Invoker/Invokee Relationship

Four levels of relationship have been identified, and they have been defined to be mutually exclusive levels. The reason behind this decision is to keep the distinct value and meaning, if any, of the different levels from being distorted or noised by each other. The following lists and formally defines the four relationship levels between the enclosing classes of the invoking method (class $c_1$) and the enclosing classes of the invoked method (class $c_2$):

- **ToSibling:** Classes are said to be siblings when they extend/implement the same class/interface. This level of relationships can be formally defined as

follows:

$$ToSibling(c_1, c_2) = \{\exists c \in C \mid (\, Super(\, c_1, c) \wedge Super(c_2, c)\,) \\ \vee\, (\, Interface(c_1, c) \wedge Interface(c_2, c)\,)\} \tag{3.5}$$

- **ToAncestor:** If the predicate $Generalize(c_1, c_2)$ holds true, class $c_2$ is said to be an ancestor for the class $c_1$ (i.e. the predicate $ToAncestor(c_1, c_2)$ holds true). Note the difference between the $ToAncestor()$ predicate defined here and the $Ancestor()$ predicate defined in the previous section.

- **ToRelative:** The two classes $c_1$ and $c_2$ are said to be relatives if they share the same super class with which they both have a generalization relationship, with the exclusion of the *ToSibling* and *ToAncestor* cases to preserve the mutual exclusiveness. The following expresses this in formal terms:

$$ToRelative(c_1, c_2) = \{\exists c \in C \mid Generalize(c_1, c) \wedge Generalize(c_2, c) \\ \wedge \neg Sibling(c_1, c_2) \wedge \neg Generalize(c_1, c_2)\} \tag{3.6}$$

- **ToUnRelated:** The two classes $c_1$ and $c_2$ are said to be unrelated when none of the above level of relationships applies.

$$ToUnRelated(c_1, c_2) = \{\neg ToSibling(c_1, c_2) \wedge \neg ToAncestor(c_1, c_2) \\ \wedge \neg ToRelative(c_1, c_2)\} \tag{3.7}$$

These levels can, in fact, have a technical effect on the execution of method invocations. For example, while method invocations to an ancestor class may actually be polymorphically executed by objects of any of its descendant classes, including the invoking class itself, the same does not apply in cases with different invoker/invokee relationships.

### 3.3.2 Method Invocation Types

Method invocations differ in four different ways. They can differ in the frequency of their invocation, whether or not the invocation is addressed to an abstract method, whether or not the invocation is a delegation and the location where the method is invoked. The following discusses these different ways with more details:

- **Single/Iterative:** A method invocation to the method $m_2$ is said to be iteratively executed if it is implemented inside a loop in the method $m_1$, and in such a case the predicate $IMI(m_1, m_2)$ (i.e. Iterative Method Invocation) holds true. If the method invocation is not implemented inside a loop, it will obviously be executed only once, and in such a case, the predicate $SMI(m_1, m_2)$ (i.e. Single Method Invocation) holds true.

- **Abstract/Concrete:** The invoked method can be an abstract or concrete method. Invoking an abstract method means that the implementation that will eventually execute the invocation will be determined polymorphically at run-time.

- **Delegation/Not-Delegation:** If the invoking and invoked methods have identical names, the method invocation is called *delegation*, which indicates that the service implemented by the invoking method may be (partially) fulfilled by the invoked method. Although *delegation* can in fact be implemented by invoking methods with different names, such cases cannot be recovered automatically [158]. So, the predicate $SameName(m_1, m_2)$ suffice to distinguish method delegations from other method invocations.

- **In New/Updating Method:** In a class $c$, method invocations can be implemented inside a *new* method $m \in M_{NEW}(c)$, which indicates that the method invocations are used by the enclosing class $c$ to implement a newly added

service. Otherwise, method invocations can be implemented inside an *updating* method $m \in M_{OVR}(c)$ or $m \in M_{IMM}(c)$, which means that the method invocations are used to implement an already declared service.

### 3.3.3 Method Invocation Features

Two options are possible for each of the four method invocation types above (e.g. `true/false` *Abstract*() predicate), which produces 16 (i.e. $2^4$) different method invocation features capturing all the possible combinations. A method invocation inside a method $m \in M_{NEW}(c)$, for example, can be iterative abstract delegation, iterative concrete delegation or iterative abstract invocation. These 16 method invocation features will be calculated for each of the four invoker/invokee relationship levels, which makes the total number of the behavioural features 64.

Recalling the aim for the feature set to provide as much design information as possible, which is emphasised by using the *degree* term, method invocation information will not be captured in a binary form as it is mostly the case in existing DP recognition approaches. The proposed features will instead capture this information as a count of the number of occurrences. One possible way of doing so is to count the number of all method invocations from one class to the other. This may, however, lead to features with distorted values. If, for example, class $c_1$ has 10 methods each of which has a method invocation to a method in class $c_2$, the relevant feature will have the same value even when all the method invocations are implemented in a single method while the others have none. A similar distortion problem is noted in [28] in support of using the number of invoked methods instead of the number of method invocations when measuring the strength of coupling between classes.

An alternative approach is to count the number of methods in class $c_1$ that have a method invocation to a class $c_2$ method. This approach is the one adopted because

it gives information about how well connected are the services implemented by the two classes (i.e. how many of the services implemented in class $c_1$ rely on services provided by class $c_2$). This is also the same counting approach as the one used in the Columbus filtering phase [63] and the three threshold rules in [43], as discussed in the previous chapter.

Each of the invoker/invokee relationship levels will be represented by one *degree* type, which groups together the 16 features calculated for the corresponding level. The following list gives examples for how some of the features included in each *degree* type may be relevant to some DPs. In cases when a proposed feature happens to resemble some similarity to existing work in the field, the identified similarity will be noted and contrasted. One feature will also be formally defined for each *degree* type, and other features follow the same template.

- **Degree of ToSibling Invocation:** This set of features includes a feature that captures non-iterative concrete method delegations which are implemented in updating methods. This feature[2] may be relevant to the Proxy DP, and it is formally defined for classes $c_1$ and $c_2 \in C$ as follows:

$$SSCDU(c_1, c_2) = |\{m_1 | m_1 \in (M_{OVR}(c_1) \cup M_{IMM}(c_1)) \wedge ToSibling(c1, c2) \\ \wedge \exists m_2 \in c_2 : SMI(m_1, m_2) \wedge \neg Abstract(m_2) \wedge SameName(m_1, m_2)\}| \tag{3.8}$$

  This feature as well as its corresponding *invocation* (i.e. not *delegation*) one, are respectively similar to the *DelegateInLimitedFamily* and *RedirectInLimitedFamily* Elemental Design Patterns (EDP) [116]. The EDPs differ, however, in being representable by binary features, which means they do not carry as much design information as the proposed features.

[2] toSibling-Single-Concrete-Delegations-inUpdating feature.

- **Degree of ToAncestor Invocation:** The feature in this set which captures iterative abstract method delegations that are implemented in updating methods may be relevant to the Composite DP. The corresponding *non-iterative* feature may, however, be relevant to the Decorator DP. The former feature[3] is formally defined as follows:

$$AIADU(c_1, c_2) = |\{m_1 | m_1 \in (M_{OVR}(c_1) \cup M_{IMM}(c_1)) \wedge ToAncestor(c1, c2) \\ \wedge \exists m_2 \in c_2 : IMI(m_1, m_2) \wedge Abstract(m_2) \wedge SameName(m_1, m_2)\}| \tag{3.9}$$

  The *iterative* feature above is similar to the *Multiple Redirections in Family* DP clue [66], and both features are similar to the *DelegateInFamily* EDP. DP clues differ, like it is the case with the EDPs, in being representable by binary features.

- **Degree of ToRelative Invocation:** The feature in this set which captures non-iterative concrete method delegations that are implemented in updating methods may be relevant to a variant of the Proxy DP in which the Proxy class and the RealSubject class are not siblings (e.g. they have an indirect generalization relationship with their super Subject class). This feature[4] is formally defined as follows:

$$RSCDU(c_1, c_2) = |\{m_1 | m_1 \in (M_{OVR}(c_1) \cup M_{IMM}(c_1)) \wedge ToRelative(c1, c2) \\ \wedge \exists m_2 \in c_2 : SMI(m_1, m_2) \wedge \neg Abstract(m_2) \wedge SameName(m_1, m_2)\}| \tag{3.10}$$

- **Degree of ToUnRelated Invocation:** The feature in this set which captures non-iterative abstract method invocations that are implemented in updating

[3] toAncestor-Iterative-Abstract-Delegations-inUpdating feature.
[4] toRelative-Single-Concrete-Delegations-inUpdating feature.

methods may be relevant to the Adapter DP. The other feature in this set that captures non-iterative abstract method invocations that are implemented in new methods may, however, be relevant to the Visitor DP, and its corresponding *iterative* feature may be relevant for the Observer DP. The latter feature[5] is formally defined as follows:

$$\begin{aligned} UIAIN(c_1, c_2) = |\{m_1 | m_1 \in M_{NEW}(c_1) \wedge ToUnRelated(c_1, c_2) \\ \wedge \exists m_2 \in c_2 : IMI(m_1, m_2) \wedge Abstract(m_2) \wedge \neg SameName(m_1, m_2)\}| \end{aligned} \tag{3.11}$$

The feature relevant to the Adapter DP above is similar to the *Adapter Method* DP clue although, as mentioned earlier, the clue does not carry as much design information.

Out of the 64 features produced to cover all the levels/types variations, only a few have been found to have obvious or immediate relevance to a DP in a way that could have been deduced from their descriptions. Other method invocation features may provide important design information that could, according to the descriptions, appear to be irrelevant. Some other features may be relevant to certain DP variants, or not relevant at all to any of the GoF set of DPs.

## 3.4 Normalisation

Although the features proposed in the previous two sections do represent a rich set of features that already hold traces of intent-related information, their values are highly dependent on the scale of the measured DP instances, which varies greatly according to the specific design problems being solved. Such variations may obscure any existing traces of intent information. So, in Section 3.4.2, an alternative normalisation based feature calculation method is proposed to solve this problem.

[5] toUnrelated-Iterative-Abstract-Invocation-inNew feature.

However, the problem of not taking the scale context into account is first discussed with more details in the following subsection.

### 3.4.1 Effect of Context Variations

If the context of DP instances is not taken into account when calculating the features by which they are represented, the calculated feature values will be highly influenced by and dependant on the particular size of different DP instances. For example, two classes that have only *updating* methods may have different values for the $UPM$ feature depending on the total number of methods in each one, although they both are identical in terms of the measured property. Similarly, for two sibling classes $c_1$ and $c_2$, if the class $c_1$ delegates the execution of all of its *updating* methods to its $c_2$ sibling, the value of the $SSCDU(c_1, c_2)$ feature will depend on the total number of methods in class $c_1$.

The two aforementioned features can also have the same value for classes in which the measured property differs. If, for example, the classes $c_1$ and $c_2$ have two *updating* methods out of two and two *updating* methods out of twenty, respectively, both classes will have the same $UPM$ value although they are different in terms of the measured property. Only the class $c_1$ can be said to be *intended* to update super class methods, and this difference between the two classes will not reflected on the value of $UPM$ feature. This can only be inferred by comparing the value of the $UPM$ feature with the total number of declared methods in the class. This shows that if the size context is not considered, the feature values may not be as meaningful on their own.

This problem is basically caused by measuring the absolute numbers instead of the proportions they represent. Measuring proportions leads to values that are meaningful on their own in way that may provide better information for *intent* inference. A solution to this problem is discussed in the following subsection.

### 3.4.2 Context-Based Normalisation

As hinted in the previous subsection, the solution to the problem described above lies on using proportions instead of absolute numbers. The features, as they are calculated in the previous two sections, can simply be proportionalised (i.e. normalised) by their relevant denominators to produce a new set of normalised features. The normalised features will measure, for example, *how much* (instead of *how many*) of the services implemented in class $c_1$ rely on services provided by class $c_2$. Such normalised features appropriately capture the measured properties in a way that may enrich each individual feature with a better trace of *intent* information, and different combinations of these features should provide valuable information about the *intent* of potential DP instances.

#### 3.4.2.1 Normalisation Denominators

The largest value possible for each feature in class $c$ is used as a denominator to normalise its values. Most of the features that count numbers of methods can be normalised by using the number of declared methods (i.e. $M_{DEC}(c)$) as a denominator. The values of the $IHM$ feature is, however, normalised by using $(M_{IHM}(c) + M_{DEC}(c) - PriM(c))$[6]. The number of private methods is subtracted from this denominator because they cannot implement or override inherited methods. The same applies for the denominators used for the behavioural features that count the number of method invocations implemented in *updating* methods.

The features that count numbers of attributes can be normalised by the total number of declared attributes (i.e. $A_{DEC}(c)$). The two *Degree of Ancestry* metrics (i.e. $DIT$ and $CLD$) can be normalised by their sum, which can then give information

[6] $PriM(c)$ donates the number of private methods. The normalised IHM feature is similar to the Measure of Functional Abstraction (*MFA*) metric, which is introduced in [18] as a part of a quality evaluation model. The *MFA* metric, however, does not exclude the private methods from the denominator.

about how far a class is from the root and leaf of the hierarchy in which it resides. A class that is close to the top of the hierarchy will have values close to 0 and 1 for the normalised $DIT$ and $CLD$, respectively, regardless of the number of levels in the hierarchy. The binary features (e.g. association features) are the only ones that are not, and obviously do not need to be, normalised.

#### 3.4.2.2 Normalisation Calculation

The normalised features can simply be calculated by dividing the absolute values of their unnormalised counterparts by the relevant denominators. The normalised counterpart of the $PubM$ feature in Eq. (3.1), for example, can be calculated as follows:

$$PubM^*(c) = \begin{cases} 0 & |M_{DEC}(c)| = 0 \\ \frac{PubM(c)}{|M_{DEC}(c)|} & Otherwise \end{cases} \tag{3.12}$$

Both the normalised and the unnormalised features will be used to create two different dataset versions for each classifier to be trained, and their classification performances will be compared in Chapter 6. The denominators $M_{DEC}(c)$ and $A_{DEC}(c)$ will be added as separate features to the unnormalised version of the dataset since their interaction with other features may hold valuable information as explained earlier.

## 3.5 Conclusions

A large and information-rich feature set of 97 structural and behavioural features has been introduced in this chapter. In order to avoid being limited by subjective presumptions and personal interpretations of DPs, the set has been designed to cover and measure the aspects and properties of software design not that of a certain set of DPs. All features in the set are mechanically and unambiguously

recoverable, including the novel set of association relationship features, which has been introduced specifically for this purpose.

The set is intended to be a global feature set from which different subsets can be objectively selected for different DPs, which fills a gap in the literature as no such set currently exist. Besides the default feature calculation method, an alternative normalisation-based calculation method was also proposed with the aim of providing better information for *intent* inference. The objectively selected feature subsets as well as their different calculation methods, together with complex machine learning based models, should be able to effectively tackle the problem of DP recognition, including the challenging problem of recognizing DP variant as well as the inference of potential instances' intents.

The proposed set of features can be used in different ways, depending on the design of the DP recognition system in which they are used. They can be used, for example, all together as a single global set of features. They can also be split into different global subsets that correspond to different phases in the recognition process (e.g. intra/inter class phases or structural/behavioural phases). The way in which they are going to be used in this thesis will be explained in Chapter 6.

# Chapter 4

# Quality Features

Many OO metrics have been proposed and empirically validated as quality evaluation measures [136] [55]. However, there is still a limited number of empirical studies that use these metrics to test the presumed impact of DPs on software quality. Moreover, in the studies in which they have been used, their values have been inappropriately combined, improperly analysed and some anomalous metrics have also been used, as discussed in Chapter 2 (Section 2.3.3). The aim of this chapter is to contribute to filling this gap by preparing a set of quality metrics that are appropriate for the aforementioned purpose. Some of the metrics to be used need only to be formally defined since they are usually defined in an informal and ambiguous way [27] [28], and any ambiguity will be resolved in favour of the purpose of their use in this thesis. Slight adaptations to the definition of some metrics will also be required to appropriately deal with cases in which they cannot be calculated. Novel metrics will be introduced to cover coupling quality aspects that are particularly relevant to DPs but not covered by existing metrics.

The first three sections of this chapter discuss the main quality aspects (i.e. cohesion, coupling and complexity), respectively. The discussions include brief definitions for these aspects, their expected relationships with DPs as well as the

metrics used to measure them. Metrics that take the scale-context into account are preferred and used whenever possible. For the intra-class quality attributes (i.e. cohesion and complexity), such metrics exist in which class size measures are used as normalizers. However, for the intra-class quality attribute (i.e. coupling), a new approach is introduced in Section 4.4 for the normalization of the coupling metrics with regards to the size of the software system. Section 4.5 discusses the outcomes of this chapter and how the work presented here relates to the rest of this thesis.

## 4.1 Cohesion

The cohesion of a class can be defined as the consistency among the elements (i.e. methods and attributes) of the class [38]. This consistency is measured as "the degree to which the methods and attributes of a class belong together" [27, page 66]. The methods and attributes of a cohesive class are bundled together to perform the same function. This is called *functional cohesion*, which is the most desirable form of cohesion, as opposed to *coincidental cohesion*, which is the least desirable form [34]. It is hypothesized that cohesive classes are easier to maintain and less fault-prone [27].

### 4.1.1 Design Patterns and Cohesion

One of the design principles to which DPs conform is the single responsibility principle [8]. In the Composite DP, for example, the responsibility of composing and iterating through a structure of objects is assigned to the Composite role while the responsibility of defining the behaviour of individual objects is assigned to the Leaf role. Although the concept of cohesion is more general than the principle of single responsibility, classes that conform to this principle tend to be more cohesive [69].

### 4.1.2 Cohesion Metrics

Four metrics will be used to measure the cohesion of classes, and they complement each other in a way that covers different possible cases and situations. The four metrics are slightly modified versions of existing ones, and they are defined and discussed in the following subsections.

#### 4.1.2.1 Attribute based Method Cohesion ($AMC$)

Chidamber and Kemerer proposed the well-known $LCOM$ cohesion metric, which measures how cohesive the methods of a class are based on their use of common attributes [38]. Two problems have, however, been identified in this metric in [145]: (1) it produces the same value for classes with different cohesion, and (2) there is no guideline as to how to interpret the values produced. So, a variation of this metric was proposed in [145] to solve these problems by measuring cohesion with a fraction based metric, which was named $LCOM^*$. The $LCOM^*$ metric is normalized to the range $[0, 1]$ only if each attribute in the measured class is accessed by at least one method. This is considered to be an anomaly[1] by Briand *et al.* who have resolved it by redefining the metric into a conceptually similar one, which was defined for a class $c \in C$ as follows [27]:

$$NewCoh(c) = \begin{cases} 0 & if\ |M_{DEC}(c)|=0\ OR\ |A_{DEC}(c)|=0 \\ \\ \frac{\sum_{i=1}^{|A_{DEC}(c)|}\mu(a_i)}{|M_{DEC}(c)|\times|A_{DEC}(c)|} & Otherwise \end{cases} \tag{4.1}$$

where $a_i$ is the $i^{th}$ element in the $A_{DEC}(c)$ set, and $\mu(a_i)$ is the number of methods in the $M_{DEC}(c)$ set that use the attribute $a_i$. The higher the value of the $NewCoh$

[1] Despite these anomalies and problems, the original LCOM metric as well as its aforementioned variation have been used by existing studies on the impact of DPs on cohesion, as discussed in Chapter 2 (Section 2.3.3).

metric for a class, the more cohesive the class is. However, there are still two problems with this metric. First, a perfectly cohesive abstract class would get a low value, or even zero if all of its methods are abstract, simply because abstract methods cannot be using any attribute. The second problem is, in cases when a class has no methods/attributes, the value zero will be assigned to this metric, which indicates a terrible cohesion although it may not be necessarily the case. The inability to calculate a value for this metric in a class, due to the division-by-zero problem, does not imply that the class is completely incohesive.

In order to solve these two problems, the definition of the *NewCoh* metric needs to be adapted to assign the value of 0.5, as a neutral value, to cases in which the metric value is not calculable, and to deal with cases in which the measured class is abstract as incalculable cases. Applying these changes to Eq. (4.1) produces the definition of the $AMC$ metric used in this thesis, which is defined as follows:

$$AMC(c) = \begin{cases} 0.5 & if\ |M_{DEC}(c)|=0\ OR\ |A_{DEC}(c)|=0 \\ & OR\ \exists m \in M_{DEC}(c):\ Abstract(m) \\ \frac{\sum_{i=1}^{|A_{DEC}(c)|} \mu(a_i)}{|M_{DEC}(c)|\times|A_{DEC}(c)|} & Otherwise \end{cases} \quad (4.2)$$

The problem of having cases in which the value of a feature cannot be calculated can be seen as a missing value problem, which is a common problem in most real world datasets [167]. Assigning a single global constant to such cases, as done here, is one of the approaches used to solve this problem [96].

#### 4.1.2.2 Connectivity based Method Cohesion ($CMC$)

Motivated by several anomalies identified for the $LCOM$ metric, including its dependence on the number of methods in the measured class, Hitz and Montazeri proposed a method-connectivity based cohesion measure, in which two methods

are considered to be connected if they use the same attribute, or if one of them calls the other [89]. This definition of *connectivity* allow the metric to account for cases in which methods use attributes indirectly through other (e.g. access) methods, or even not using them at all[2], which are not covered by the $AMC$ metric discussed above. However, although this *connectivity* based measure produces values in the range $[0, 1]$, it does so under an assumption without which it can produce negative values. So, to resolve this problem, the metric was redefined in [27] as follows:

$$Co'(c) = 2 \times \frac{|E|}{|V| \times (|V| - 1)} \tag{4.3}$$

where $|V|$ is the number of declared methods (vertices) in a class and $|E|$ is the number of connections (edges) between the methods. This definition also needs to be adapted in a similar way to that described for the $AMC$ metric above for the same reason. However, since this *connectivity* based metric does not rely on the existence of any attribute, the condition for assigning the neutral value is changed accordingly. Also, if a class has a single method, the method can be said to be connected to itself, which makes the class a perfectly cohesive class from the perspective of this metric, and hence the value 1 is assigned in such cases. Applying these changes to Eq. (4.3) produces the definition of the $CMC$ metric, which is defined as follows:

$$CMC(c) = \begin{cases} 0.5 & if\ |M_{DEC}(c)| = 0\ OR\ \exists m \in M_{DEC}(c):\ Abstract(m) \\ 1 & if\ |M_{DEC}(c)| = 1 \\ 2 \times \frac{|E|}{|V| \times (|V|-1)} & Otherwise \end{cases} \tag{4.4}$$

[2]This is relevant for "many practical cases in which methods exist which do not at all access instance variables (neither directly nor via mere access methods) but are coded entirely in terms of other (more basic) methods of their class" [89, page 9].

#### 4.1.2.3 Parameter based Method Cohesion ($PMC$)

This metric measures the cohesion of a class by measuring the similarity of its methods based on their use of common parameter types. It is adopted from [18][3] in which it was proposed to measure cohesion during the design phase, which means that it does not rely on the existence of any implementation information. Using such a metric is particularly useful for measuring the cohesion of abstract classes, which was not possible by using the first two metrics. The original definition of this metric needs also to be adapted in a similar way to that described for the $AMC$ metric above for the same reason. The $PMC$ metric is defined as follows:

$$PMC(c) = \begin{cases} 0.5 & if\ |PAR(M_{DEC}(c))| = 0 \\ \frac{\sum_{i=1}^{|M_{DEC}(c)|} |PAR(m_i) \cap PAR(\ M_{DEC}(c)\ )|}{|M_{DEC}(c)| \times |PAR(\ M_{DEC}(c)\ )|} & Otherwise \end{cases} \tag{4.5}$$

where $m_i$ is the $i^{th}$ element of the $M_{DEC}(c)$ set, and $PAR(\ m\ )$ is the set of unique parameter types for the set of methods $m$.

#### 4.1.2.4 All Attribute based Cohesion ($AAC$)

This metric measures the cohesion of a class with more emphasis on the set of attributes available (i.e. declared and inherited) in the class. The metric is in fact similar to the $AMC$ metric above, but it differs in the inclusion of inherited attributes. The inclusion of inherited attributes distinguishes this cohesion metric from all the previous ones, and it is formally defined as follows:

$$AAC(c) = \begin{cases} 0.5 & if\ |M_{DEC}(c)| = 0\ OR\ |A(c)| = 0 \\ & OR\ \exists m \in M_{DEC}(c):\ Abstract(m) \\ \frac{\sum_{i=1}^{|M_{DEC}(c)|} \mu(m_i)}{|M_{DEC}(c)| \cdot |A(c)|} & Otherwise \end{cases} \tag{4.6}$$

[3]The metric name in this reference is $CAM$ (Cohesion Among Methods).

where $A(c)$ is the set of declared and inherited attributes in the class $c$, $m_i$ is the $i^{th}$ element in the $M_{DEC}(c)$ set, and $\mu(m_i)$ is the number of attributes in the $A(c)$ set that are used by the method $m_i$. A similar metric has been used in [99] but, again, its definition is adapted in a similar way to that described for the $AMC$ metric above for the same reason.

## 4.2 Complexity

The complexity can be defined loosely as the characteristic of a software system that expends resources in order to build, understand and maintain the system [145]. Given this broad definition, it can be said that cohesion and coupling are essentially complexity aspects. The term *complexity* has, in fact, been used as an umbrella term for the aforementioned quality aspects in the literature [38]. However, these two aspects represent different types of complexity, and as classified in [145], coupling is an inter-class complexity aspect while cohesion is an intra-class complexity aspect. Although the complexity aspect discussed in this section is also an intra-class complexity aspect, it refers to and specifically covers the procedural complexity, which is associated primarily with the logical structure of the source code [156]. Metrics representing this complexity aspect have been used as predictors for the maintainability and fault-proneness of software classes [55] [136].

### 4.2.1 Design Patterns and Complexity

The design solutions suggested by DPs embody best practices in decomposing complex services and components into (less complex) cooperating parts [34]. The decomposed parts (i.e. classes) are usually assigned a single responsibility (given the single responsibility principle mentioned earlier), which should result in classes with relatively low complexity. Also, most DPs employ the polymorphism mecha-

nism which enables implementing different implementation alternatives in multiple classes of the same type instead of implementing them in a single complex class with a cascade of `if-then-else` or `switch/case` statements [8] [145, page 3].

### 4.2.2 Complexity Metrics

Metrics for measuring the procedural complexity have been initially proposed for the procedural programming paradigm and were later adopted for the OO paradigm. According to [136] and [145], the best known and the most frequently used among them are McCabe's cyclomatic complexity metrics [117] and Halstead's complexity metrics. However, the Halstead's complexity metrics are not used in this thesis due to the fact that they have been criticized as being "an example of confused and inadequate measurement" [62, Page 249]. Several other criticisms have also been discussed in [145, pages 90-91].

The cyclomatic complexity is basically a measure of the linearly independent paths through the control flow graph (G) of a piece of code, each node of which corresponds to a sequentially-executed block of statements and each edge corresponds to a branch generated by a control (e.g. `if`) statement. The code complexity can be calculated based on the number of decision points ($d$) as follows [117]:

$$CyclomaticComplexity(G) = d + 1 \tag{4.7}$$

The cyclomatic complexity of an OO class can be calculated as the sum of the complexities of its methods. This is a version of the Weighted Method per Class ($WMC$ [38]) metric in which methods are weighted by their cyclomatic complexity [109]. This version can be formally defined for a class $c$ as follows:

$$WMC(c) = \sum_{i=1}^{|M_{DEC}(c)|} CC(m_i) \tag{4.8}$$

where $CC(m_i)$ is the cyclomatic complexity of the $i^{th}$ method in the $M_{DEC}(c)$ set. This metric does not, however, account for the distribution of the complexity within classes, which needs to be measured to distinguish classes with the same overall complexity but distributed differently between their methods. Three complexity metrics will be used for this purpose: maximum, average and the standard deviation of method complexities[4], which are respectively defined as follows:

$$WMCmax(c) = \max_{i=1}^{|M_{DEC}(c)|} CC(m_i) \tag{4.9}$$

$$WMCavg(c) = \frac{WMC(c)}{|M_{DEC}(c)|} \tag{4.10}$$

$$WMCstd(c) = \sqrt{\frac{\sum_{i=1}^{|M_{DEC}(c)|} (CC(m_i) - WMCavg(c))^2}{|M_{DEC}(c)|}} \tag{4.11}$$

Participant classes of a DP can, for example, have similar overall level of complexity (as measured by the $WMC$ metric) as other classes but better distribution of complexity within classes (as measured by the $WMCavg$ and $WMCstd$ metrics).

## 4.3 Coupling

The degree to which software classes rely and depend on each other is known as coupling. Two class are said to be coupled if, for example, one of them invokes a method in the other or use instance variables of its type [38]. The strength of the relationships established between different classes in a software system determines how coupled the system is [153]. In a tightly coupled software system, individual classes cannot be changed without properly understanding, and potentially chang-

[4]The maximum, average and the standard deviation of method complexities have been used in previous studies on software quality evaluation (e.g. [127] and [94]).

ing, many other classes in the system. So, such systems are hard to maintain and they are also error-prone systems [28] [71].

### 4.3.1 Design Patterns and Coupling

Most of the GoF DPs "aim at reducing coupling and increasing flexibility within systems" [134, page 1134], and to achieve this aim, the use of *program to an interface not an implementation* design principle, which promotes loose coupling, represents a common theme in this set of DPs [71]. In the Observer DP, for example, subjects are loosely coupled to their observers in a way that they can both change independently from each other. Also, implementing the Command DP helps to decouple the class that requests the execution of an operation from the one that actually executes it.

### 4.3.2 Coupling Metrics

Coupling is a relatively complex quality aspect as there are many different mechanisms and ways in which coupling relationships can be established and measured [28]. In order to appropriately analyse the impact of DPs on coupling, it is necessary to identify the types of coupling that are relevant to DPs, and make sure that the impact, if any, on each of these types is measured by the set of metrics used. These types, or dimensions on which coupling relationships vary, are discussed in the following subsection. Then, in Section 4.3.2.2, other coupling issues are discussed and resolved in order to enable the formal definition of the metrics to be defined to measure the DP impact on the identified coupling types, and the metrics are finally defined in Section 4.3.2.3.

#### 4.3.2.1 Coupling Dimensions

The relevant types of coupling relationships differ according to the goal of measurement. For example, while attribute referencing, as a mechanism by which

coupling can be constituted, has been found in [30] to be a relevant coupling type for the purpose of ripple effect[5] analysis, it would not be the case if the purpose is to perform control flow analysis because such type of coupling does not have an impact on the flow of control [28]. Therefore, a fundamental step towards studying the coupling impact of DPs is to identify the dimensions on which coupling relationships vary that are relevant to DPs. The relevant types can, then, be specified as different combinations of the variations in these dimensions. The following lists and describes the three relevant dimensions identified:

1. **Relationship Direction:** a class can have *export* coupling relationships, in which other classes depend on the class in question, and vice versa for the *import* coupling relationships. While the import coupling gives an insight on whether or not DP instances deliver the benefits expected from the implemented DPs (e.g. do Subjects in Observer DP instances have low import coupling?), the export coupling gives an insight on how well they are used in the systems in which they are implemented (e.g. Composites in Composite DP instances should ideally have low export coupling.).

2. **Class Type:** Coupling relationships can be *abstract* or *concrete* couplings. An abstract/concrete coupling relationship is the one in which the class at the exporting end of the relationship is an abstract/concrete class, respectively. This dimension is closely relevant to DPs given their aim and the common thematic design principle they implement to achieve it, as discussed earlier.

3. **Level of Abstraction:** coupling relationships can be established at the *design* or *implementation* levels of abstraction. Coupling relationships that are established through class attributes, method parameters and method return types are *design*-level couplings while, on the other hand, those established by

[5]The ripple effect is the effect of change and error propagation between different parts of a software system [153].

method invocations or attribute referencing are *implementation*-level couplings. It can be said that coupling relationships at the design level are more likely to be influenced by DPs than those established at the implementation level since DPs are defined at the design level.

Specifying a coupling type for each possible combination of the variations in the dimensions above (e.g. *abstract design import* coupling and *abstract implementation import* coupling), as done with the coupling dimensions identified in [29], does not suit the goal of measurement in this thesis as it would lead to segmented and fragmented measures of the DP impact. Also, export coupling relationships would probably be affected less, if at all, by design decisions made during the instantiation of DPs as they are likely be affected more by design decisions made elsewhere in the system. So, details of the export coupling relationships of DP participant classes do not matter as much as the total number of such relationships.

With these two factors in mind, four import-coupling types will be specified (i.e. *abstract* coupling, *concrete* coupling, *design* coupling and *implementation* coupling) as well as one export coupling type (i.e. *all-export* coupling). One metric will be defined to measure the DP impact on each of these coupling types. The metric to be defined for the *design* coupling type is similar to the DCC (Direct Class Coupling) metric in [18], which also targets import coupling at the design level, although it apparently excludes couplings established through method return types. Also, the metric to be defined for the *implementation* coupling is similar to the well-known CBO (Coupling Between Object classes) metric [38], but the CBO metric differs in measuring both import and export coupling. No coupling metrics exist that distinguish between *abstract* and *concrete* coupling.

In order to formally define the coupling metrics, there are still some decisions to be made with regards to some other coupling issues, which are discussed in the following subsection.

#### 4.3.2.2 Other Coupling Issues

There are three issues that need to be discussed and resolved, and the decisions to be made with regard to resolving them have to be made with respect to the intended application of the metrics [28]. The following discusses these issues and justify the decisions made.

The first issue is whether or not inheritance-based coupling (e.g. the invocation of an inherited method) should be counted. Such case of couplings result from using inheritance as a reuse mechanism, which is something that has been argued against in the GoF book as it breaks the encapsulation of, and increases the dependency on, the reused classes. A better reuse mechanism alternative is object composition as it avoids the aforementioned problems and also enables the flexibility of changing the system's behaviour at run-time. The *favour object composition over class inheritance* is, in fact, one of the design principles considered by the authors of the GoF DPs [71]. So, inheritance and non-inheritance based couplings will be indistinguishably counted.

The ambiguity that arises with the invocation of polymorphic or inherited methods in other classes is the second issue to be resolved. If an inherited method in class ($c1$) is invoked by class ($c2$), should this invocation contributes to the coupling between the two classes? If, instead, the invocation is statically addressed to a method in a superclass of $c1$ that is overridden in $c1$, should the class $c2$ also be coupled with $c1$ based on the premise that the overriding method in $c1$ can be polymorphically invoked? To resolve these two questions, the following rule is set:

> Coupling is constituted between the class in which the invocation is made and *only* the static class-type of the variable used to access the invoked method, even if the invoked method is an inherited one.

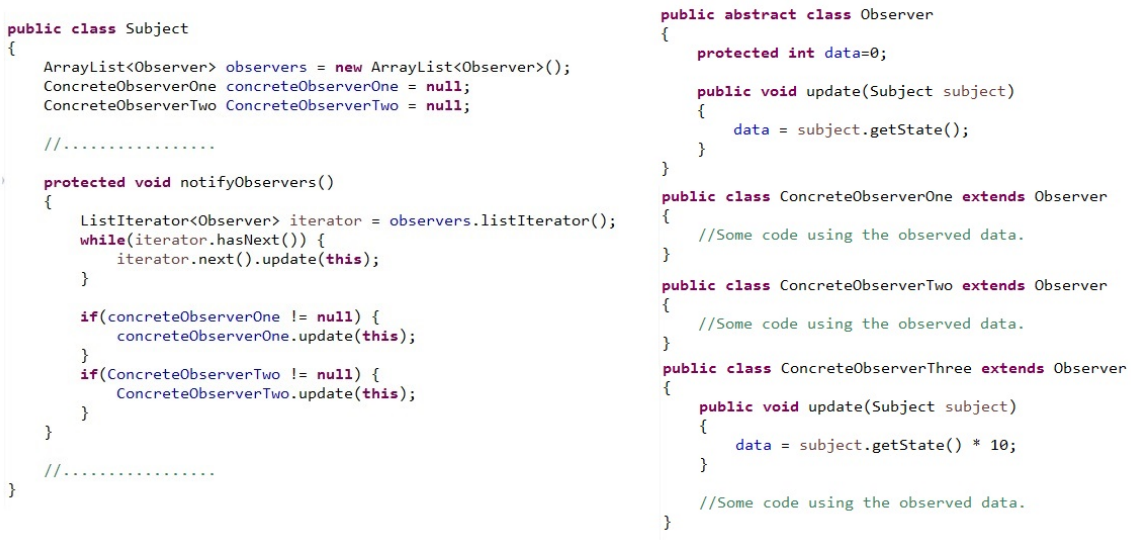
This rule explicitly resolves the invocation of inherited methods issue. As to the issue of polymorphically invocable methods, they are not considered as suggested by the word *only* in the rule.

The rationale behind this rule can be explained as follows. Let us take, for example, the Observer instance in Fig. 4.1, which is deliberately distorted[6] to explain the rationale of the rule above. As can be seen in the figure, the `Subject` class is not only abstractly coupled to its observers through their super `Observer` class, but also coupled to two of the concrete observer classes (i.e. `ConcreteObserverOne` and `ConcreteObserverTwo`) to which the invocations of the `update()` method are statically addressed. Although such a distortion clearly increases the coupling of the `Subject` class, this increase will not be reflected on the values of coupling metrics if the invoker class (`Subject`) was coupled to the class in which the invoked method is actually implemented (`Observer`), as suggested in [58], and not to the static class-type of the variable used to access them, as stated by the rule above.

If, on the other hand, the invoker class (`Subject`) was coupled to the class in which the invoked `update()` method is overridden (`ConcreteObserverThree`), and not only to the class to which the invocation is statically addressed (`Observer`), the benefits gained from being loosely coupled to the ConcreteObservers, exactly as prescribed by the Observer DP, would be obscured.

The third issue to be resolved is to determine a suitable approach for counting the coupling connections between classes. There are mainly two approaches to measure the frequency of such connections. The first is to count the number of connection occurrences (e.g. the number of invoked methods from other classes) for each class. The problem with this approach, however, is that it can produce misleading measures since a class with 10 different connections to one another

[6]It is distorted in the sense that it is implemented in a way that violates the description and purpose of the Observer DP.



```
public class Subject
{
    ArrayList<Observer> observers = new ArrayList<Observer>();
    ConcreteObserverOne concreteObserverOne = null;
    ConcreteObserverTwo ConcreteObserverTwo = null;

    //.................

    protected void notifyObservers()
    {
        ListIterator<Observer> iterator = observers.listIterator();
        while(iterator.hasNext()) {
            iterator.next().update(this);
        }

        if(concreteObserverOne != null) {
            concreteObserverOne.update(this);
        }
        if(ConcreteObserverTwo != null) {
            ConcreteObserverTwo.update(this);
        }
    }

    //.................
}
```

```
public abstract class Observer
{
    protected int data=0;

    public void update(Subject subject)
    {
        data = subject.getState();
    }
}
public class ConcreteObserverOne extends Observer
{
    //Some code using the observed data.
}
public class ConcreteObserverTwo extends Observer
{
    //Some code using the observed data.
}
public class ConcreteObserverThree extends Observer
{
    public void update(Subject subject)
    {
        data = subject.getState() * 10;
    }

    //Some code using the observed data.
}
```

Fig. 4.1 A snippet of a distorted Observer instance.

class may have the same coupling metric value as another class that is coupled to 10 other classes [28].

The second approach is the binary approach, in which the number of connections between a pair of coupled classes does not matter as long as there exist at least one. Although this approach is criticised in [28] on the basis that it does not make use of available information that could be beneficial for assessing the maintainability and testability of classes, this is the approach adopted given that DPs generally aim at reducing the number of classes to which a class is coupled, not the number of connections between them.

#### 4.3.2.3 Metric Definitions

Having made the specification decisions required, it is now possible to formally define the coupling metrics for the five coupling types identified in Section 4.3.2.1. The metrics are defined for class $c \in C$ as follows:

**Import Abstract Coupling (IAC):**

$$\begin{aligned} IAC(c) = |\{\ d \in C - \{c\} \mid Abstract(d)\ \wedge\ (\ \ (\exists a \in A_{DEC}(c) : Type(a,d)\ ) \\ \vee\ (\ \exists p \in PAR(M_{DEC}(c)) : Type(p,d)\ ) \\ \vee\ (\ \exists r \in RET(M_{DEC}(c)) : Type(r,d)\ ) \\ \vee\ (\ \exists p \in PAR(CON(c)) : Type(p,d)\ ) \\ \vee\ (\ \exists m \in (M_{DEC}(c) \cup CON(c)) : (MI(m,d) \vee AR(m,d))\ )\ \ )\ \}| \end{aligned} \tag{4.12}$$

where $MI(m,d)$ is the predicate which is true iff the body of the $m$ method includes a method invocation to a method (declared or inherited) in class $d$, and $AR(m,d)$ is the predicate which is true iff the body of the $m$ method includes an attribute reference to an attribute (declared or inherited) in class $d$. The definition of the corresponding concrete metric (i.e. **Import Concrete Coupling (ICC)** ) follows the same template and it only requires negating the $Abstract()$ predicate in Eq. (4.12).

**Import Design Coupling (IDC):**

$$\begin{aligned} IDC(c) = |\{\ d \in C - \{c\} \mid (\ \exists a \in A_{DEC}(c) : Type(a,d)\ ) \\ \vee\ (\ \exists p \in PAR(M_{DEC}(c)) : Type(p,d)\ ) \\ \vee\ (\ \exists r \in RET(M_{DEC}(c)) : Type(r,d)\ ) \\ \vee\ (\ \exists p \in PAR(CON(c)) : Type(p,d)\ )\ \}| \end{aligned} \tag{4.13}$$

**Import Implementation Coupling (IIC):**

$$IIC(c) = |\{\ d \in C - \{c\} \mid (\ \exists m \in (M_{DEC}(c) \cup CON(c)) : (MI(m,d) \vee AR(m,d))\ )\ \}| \tag{4.14}$$

**All Export Coupling (AEC):**

$$\begin{aligned} AEC(c) = |\{\, d \in C - \{c\} \mid (\, \exists a \in A_{DEC}(d) : Type(a,c)\,) \\ \vee\, (\, \exists p \in PAR(M_{DEC}(d)) : Type(p,c)\,) \\ \vee\, (\, \exists r \in RET(M_{DEC}(d)) : Type(r,c)\,) \\ \vee\, (\, \exists p \in PAR(CON(d)) : Type(p,c)\,) \\ \vee\, (\, \exists m \in (M_{DEC}(d) \cup CON(d)) : (MI(m,c) \vee AR(m,c))\,)\,\}| \end{aligned} \tag{4.15}$$

## 4.4 Normalisation

All the metrics used for cohesion are normalised to the range $[0,1]$, and the denominators used in their calculation correspond to different class size measures. So, it can be said that the value of these metrics are calculated relatively to the class size as a contextual factor. The $WMCavg$ metric used for complexity can also be said to provide a relative complexity measure that is normalised by the number of class methods as a class size measure. The importance of using such normalised metrics, and the problems that arise with the use of non-normalised ones, have been noted during the discussion of these metrics. With regards to coupling, on the other hand, only absolute metrics with no normalization at all have been used.

Just as class size measures were used as normalisers for the metrics of the intra-class quality attributes (i.e. cohesion and complexity), the size of the software system can be used to normalise the metrics of the inter-class quality attribute (i.e. coupling). The following subsection provides more arguments for why the overall size of software systems represent a valid contextual factor for coupling metrics, and why thresholds calculated relative to this contextual factor can be appropriate normalisers. Then, Section 4.4.2 describes the research method used to model and test the size-threshold relationship, and the results are presented in Section 4.4.3.

The last Section (4.4.4) discusses how this modelled size-threshold relationship can be used to calculate relative normalisers for the coupling metrics.

### 4.4.1 Software Size as a Context

Many OO metrics (including coupling metrics) have been found to follow a power law distribution [93] [113] [147], which means that the largest value of these metrics increases as the number of classes increases in a software system [111]. So, what can be described as a high coupling metric value in one system may not be so high in another system. This can actually be intuitively expected. A class that is coupled to 5 other classes in a 10 class system, for example, is perceived differently from a class that has the same number of couplings in a 100 class system.

The influence of software size on the range of OO metrics values may be particularly true for coupling metrics. Taube-Schock *et al.* conducted a study to answer the question of whether or not high coupling is avoidable, and they concluded that it is not and that the scale of coupling metrics values is positively influenced by the number of classes in software systems [155]. Also, Chidamber and Kemerer state that "coupling between classes is an increasing function of the number of classes in the application" [38, page 487]. Neglecting the effect of software size as an important contextual factor may be an underlying cause of the accuracy problem that arises when a defect prediction model is trained by using a software defect dataset from one company to predict defects in the system of another company (i.e. cross company defect prediction [131]).

Given the arguments above, if the size of software systems is not taken into account when the impact of DPs on coupling is analysed, any conclusions drawn will probably be dependent on the particular composition of the dataset used in the analysis. If, for example, most of the DP samples in the dataset happen to be taken from large software systems while most of the baseline samples happen to be

from small ones, it may be concluded that the DP samples have significantly higher coupling values, and that the DP has a negative impact on coupling.

It may not, however, be appropriate to directly use software size as a denominator to normalise the values of the coupling metrics. This is because there is no basis for assuming that the ratio of their values with respect to the software size is an accurate model or representation of the significance of these values as quality indicators. We cannot, for example, assume that being coupled to 6 other classes in a 20 class system is as bad as being coupled to 60 other classes in a 200 class system. The significance of these values may be better evaluated by using thresholds, which represent reference points with which metric values can be objectively interpreted [108]. If thresholds objectively calculated in systems of different sizes are found to be increasing with the size of the systems in which they are calculated, such thresholds would be perfect denominators to normalise the metric values in these systems.

Although it has been suggested in [145] that thresholds should not be *absolute* but rather *relative* to some contextual factors, and it has also been suggested in [176] that thresholds derived from a small software system may not accurately evaluate the quality of larger ones, no attempt has been made to develop an approach for calculating thresholds relatively to the size of software systems. There are, in fact, generally few studies on thresholding software metrics [146]. Nevertheless, a positive size-threshold relationship has manifested itself in the few thresholds that have been calculated for coupling metrics based on different threshold calculation approaches, as can be seen in Table 4.1. The three threshold calculation approaches presented in the table (i.e. [148], [21] and [147]) have been used to calculate thresholds for different software systems of different sizes[7]. One may notice that, in most cases, the same approach produces higher thresholds for larger systems.

[7]The sizes as presented in the table are relative within the set of systems used by each approach. For example, while the large system in [21] has 174 classes, the small system in [147] has 684 classes.

Table 4.1 Thresholds calculated in small (S), medium (M), large (L), larger (Lx) systems.

| Metric Name | [148] | | | [21] | | [147] | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **S** | **M** | **L** | **S** | **L** | **S** | **M** | **L** | **Lx** |
| **Coupling Between Object classes (CBO)** | 8 | 13 | 10 | 1 | 8 | 4 | 8 | 17 | 25 |
| **Response For a Class (RFC)** | 36 | 39 | 44 | 27 | 25 | 17 | 5 | 24 | 48 |
| **Coupling Through Message passing (CTM)** | 30 | 33 | 35 | | | | | | |

In order to be able to calculate thresholds on the fly to normalize the values of the five coupling metrics introduced in the previous section, the relationship between their thresholds and the size of the system in which they are calculated (i.e. henceforth, size-threshold relationship) need to be modelled and tested. The research method used for this purpose is described in the next subsection.

### 4.4.2 Research Method

To model and test the size-threshold relationship for the five coupling metrics, a set of software quality datasets representing different software systems of various sizes need to be prepared, the preparation of which is discussed in the following subsection. Then, a threshold is calculated for each metric in every system, and the threshold calculation approach used is introduced in Section 4.4.2.2. This will produce for each metric a list of software sizes, which represent the independent variable in the size-threshold relationship, and a corresponding list of thresholds, which represent the dependent variable. The process of analysing, modelling and testing the size-threshold relationships are finally discussed in Section 4.4.2.3.

#### 4.4.2.1 Software Quality Datasets

A total of 35 software defect datasets are used, representing 35 versions of 12 open source Java systems, the sizes of which vary from 26 to 994 classes[8]. Each dataset contains a list of classes along with the number of defects found in them. The selection of fault-proneness as the external quality attribute based on which thresholds are calculated is motivated by (1) the fact that improving this attribute is a major objective for software metrics research [61], and (2) the number of available datasets for this quality attribute.

Classes in the original datasets are, however, represented by a different set of metrics, and hence the datasets need to be regenerated to replace them with the five coupling metrics. Also, in the regenerated datasets, the classes are labelled as faulty (i.e. number of defects$\geq$1) or not faulty because the model used to calculate the thresholds requires a binary dependent variable. Since the size of the systems is an important variable of interest, classes from external libraries are removed and not counted. Counting such classes when measuring the size of the systems may obscure or distort existing size-threshold relationships, if any.

#### 4.4.2.2 Threshold Calculation

The threshold calculation approaches in the literature can be divided into two groups. The first group includes approaches that relate thresholds to an external quality attribute (e.g. [21], [146] and [148]). The second group includes distribution based approaches (e.g. [6], [65] and [128]), which calculate thresholds based on the distribution of metric values by using the quantile function, for example.

The approach adopted is the one used in [146], which has originally been developed in the field of epidemiology [20]. This approach is based on an equation

[8]The datasets are obtained from two publicly available repositories: Predictor Models in Software Engineering (PROMISE) [118] and Bug Prediction Dataset [46].

derived from the univariate logistic regression model. Instead of estimating the probability of having a defect in a class based on a given metric value, as it is the case with the logistic regression model, the derived equation calculates a value (threshold) based on a given acceptable risk (probability) level. However, the approach failed in [146] to calculate proper thresholds for two of the three systems studied (i.e. generated negative thresholds). Although acknowledged, the failure was not explained nor apparently understood. It failed simply because of the arbitrarily set risk levels which happen to be less than the background risk (i.e. the risk when the metric value equals zero). So, a new systematic method for calculating the acceptable risk level will be introduced and used below.

The steps of calculating the acceptable risk level, and consequently the thresholds, are explained as follows. The general model of logistic regression is:

$$P(x) = \frac{e^{\alpha+\beta x}}{1+e^{\alpha+\beta x}} \tag{4.16}$$

where $P(x)$ is the probability (risk) of having a fault, given $x$ as a metric value, and $\alpha$ and $\beta$ are the parameters to be estimated. The risk when a metric value equals zero ($p_0$) can be calculated by the following equation:

$$p_0 = P(0) = \frac{e^{\alpha}}{1+e^{\alpha}} \tag{4.17}$$

The risk denoted by $p_0$ is the background risk. The Acceptable Risk Level ($ARL$) can then be determined based an acceptable increase to the background risk. If the acceptable increase is set to 0.15, for example, the $ARL$ can simply be calculated as:

$$P_{ARL} = p_0 + 0.15 \tag{4.18}$$

Finally, the threshold (i.e. the metric Value at the ARL) can be calculated by using the equation:

$$VARL = \frac{1}{\beta}(\ln(\frac{P_{ARL}}{1 - P_{ARL}}) - \alpha) \tag{4.19}$$

One motivation for using this particular threshold calculation approach is that it involves validating the newly introduced coupling metrics as quality indicators, which are demonstrated by the p-values obtained by fitting the logistic regression models. A model will be fit for each of the five metrics in each of the 35 systems, and a threshold will be calculated only when a statistically significant association (at p-value$<$0.05) is found between the metric and the fault probability.

Most of the datasets used here are imbalanced, as it is the case in many of the available fault prediction datasets, and this problem is often ignored in software engineering studies [139]. In logistic regression, the imbalance problem affects the intercept coefficient ($\alpha$), which may distort the background risk calculation in Eq. (4.17) as well as the threshold calculation in Eq. (4.19). Although such a problem can easily be corrected, the correction needs information about the ratio of faults in the population ($y$) and their ratio in individual datasets ($\bar{y}$). So, in order to be able to apply the correction, we need to assume that the average ratio of faults in the 35 systems (0.363) is the ratio of faults in the population. The correction equation to be applied is [100]:

$$\hat{\alpha} = \alpha - \ln((\frac{1 - y}{y})(\frac{\bar{y}}{1 - \bar{y}})) \tag{4.20}$$

Results will be presented with and without the correction, and the corrected models will be used only if they lead to producing better thresholds.

#### 4.4.2.3 Data Analysis and Model Building

Having calculated a list of thresholds and a corresponding list of software sizes for each coupling metric, it is now possible to proceed to analysing, modelling and testing the size-threshold relationships.

The size-threshold relationship is analysed by assessing the strength of the association between the two variables, which is done by using the robust Spearman rank correlation analysis. However, not all size-threshold pairs from all systems and their versions are included in the correlation analysis but only one size-threshold pair per system. In cases where a system has multiple versions, the size-threshold pair from only the latest version is used[9]. This is to avoid having a positive correlation that is due merely to quality improvement, which may enable the calculation of larger thresholds, in subsequent system versions, which are also larger in size. However, these quality "improvements are small and not usually very dramatic" [18, page 12].

Finding statistically significant correlations would show that thresholds, as objective quality evaluation reference points, do increase with software size, and that modelling this size-threshold relationship should presumably enable the calculation of *good* thresholds based only on software size. To test this assumption, univariate regression models are built to relate the independent (size) variable to the dependent (threshold) variable, and the ability of the models built to estimate *good* thresholds is evaluated based on how accurately the estimated thresholds can classify classes into faulty/non-faulty ones. The regression models are, at this stage, fitted on all size-threshold pairs from all systems/versions except all versions of one system which are reserved for testing. If a reasonable classification accuracy is achieved, all size-threshold pairs are then used in the fitting of the regression models which

[9]The latest version used varies from one metric to another since not all metrics are found to have a statistically significant relationship in all systems.

Table 4.2 Number of systems in which metrics have a significant association with defects.

| **Coupling Metric** | **Significance Count** (out of 35) |
|---|---|
| Import Abstract Coupling (IAC) | 29 |
| Import Concrete Coupling (ICC) | 31 |
| Import Design Coupling (IDC) | 27 |
| Import Implementation Coupling (IIC) | 31 |
| All Export Coupling (AEC) | 13 |

will later be used to estimate the normalization thresholds for the five coupling metrics in the DP datasets.

### 4.4.3 Results

#### 4.4.3.1 Validation of the Coupling Metrics

Table 4.2 shows the number of systems in which the new coupling metrics have been found to have a statistically significant association with class defects. It can be seen that the export coupling metric ($AEC$) has been found to have a significant association in about half the number of systems as the import metrics. This is consistent with the results reported in the literature for export coupling metrics, which show that they are, in most cases, inferior to import metrics in the prediction of class defects [136]. It can, however, be said that all of the new coupling metrics are empirically validated as quality indicators.

#### 4.4.3.2 Size-Threshold Correlation

Table 4.3 shows the correlation coefficients for the relationships between thresholds and the size of the systems in which they are calculated, before and after applying the correction in Eq. (4.20). The thresholds included in the correlation analysis are calculated with 0.15 set as the acceptable risk level increase in Eq. (4.18). The correlations have been tested for higher risk level increases (i.e. 0.20 and 0.25)

Table 4.3 Correlation coefficients before and after applying the correction.

| Coupling Metric | Degrees of Freedom | Before | | After | |
|---|---|---|---|---|---|
| | | $\rho$ | p-value | $\rho$ | p-value |
| IAC | 9 | **0.65** | **0.02886** | **0.66** | **0.02598** |
| ICC | 10 | 0.46 | 0.13095 | **0.62** | **0.03317** |
| IDC | 10 | **0.73** | **0.00736** | **0.86** | **0.00033** |
| IIC | 9 | 0.3 | 0.37008 | 0.27 | 0.41714 |
| AEC | 5 | **0.71** | **0.05>P>0.025** | **0.93** | **0.01>P>0.005** |

and similar correlation coefficients have been obtained. Three metrics have strong correlations before applying the correction, and the number increased to four after applying it, and so does the correlation coefficients, which shows the benefit of correcting the distortion caused by the imbalanced datasets. Since significantly strong correlations having been found for four out of the five coupling metrics, it can be concluded that, as far as this thesis is concerned, the size-threshold relationship does exist for coupling metrics.

#### 4.4.3.3 Modelling the Size-Threshold relationships

The size-threshold relationships, as can be seen in the scatter plots in Fig. 4.2, seem to be suitably modelled by logarithmic regression models. Similarly to the correlation analysis above, thresholds here are also calculated with 0.15 set as the acceptable risk level increase in Eq. (4.18), but only after applying the correction.

Models are first fitted on all size-threshold pairs from all systems/versions except three versions of the Apache Ivy system, in which the estimated thresholds are tested. Since all classes of any system need to be coupled with at least one other class in order to be a part of the system, a threshold of two is used whenever the estimated thresholds equal to or below one. Also, since coupling metrics can have only integer values, the estimated thresholds are rounded to the nearest integer. The thresholds are then used to classify classes into faulty (i.e. metric value$\geq$threshold)

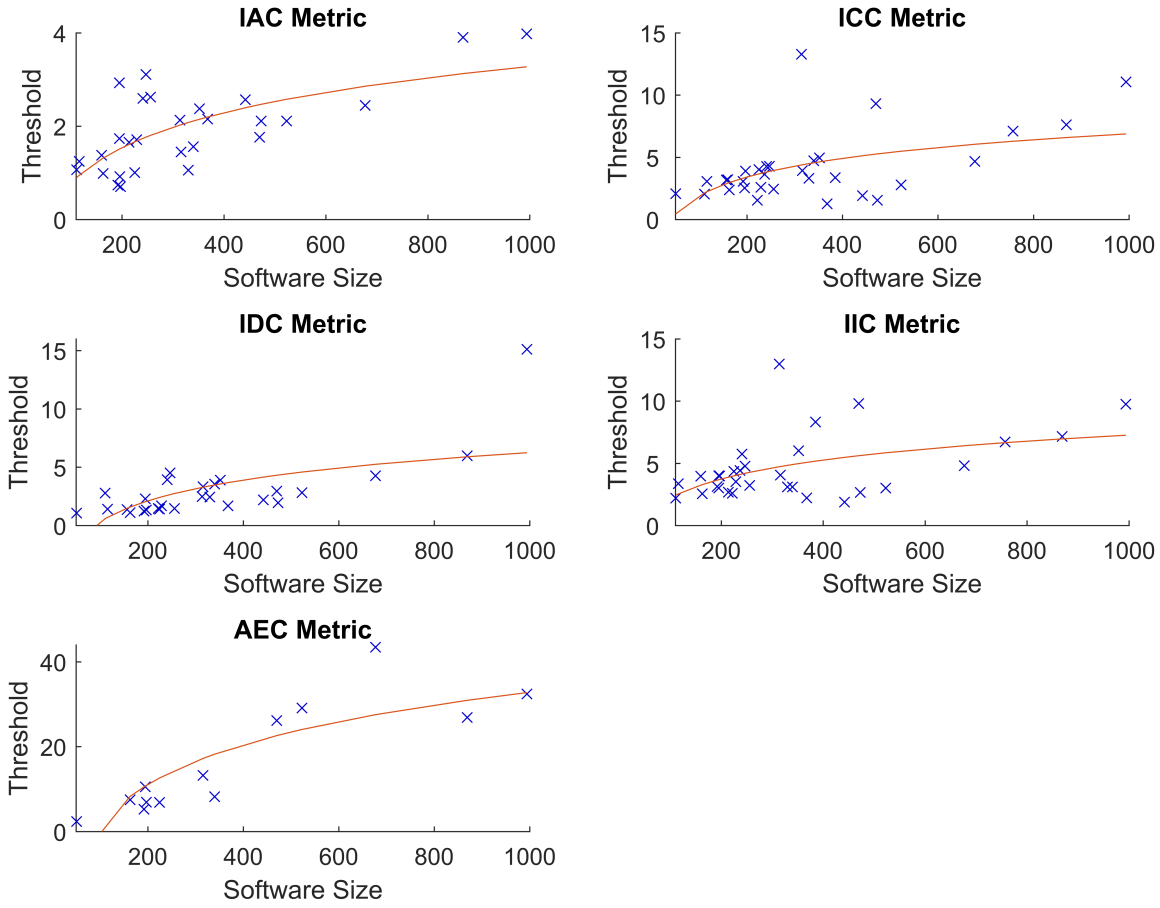



Fig. 4.2 Scatter plots for the size-threshold relationship in the coupling metrics.

or non-faulty ones. The classification accuracy is measured by using the geometric mean (g-mean), which is insensitive to the imbalance in the datasets [86].

Table 4.4 shows the classification accuracy achieved with thresholds estimated for the metrics in the three versions of the Apache Ivy system. The classification accuracy achieved with the thresholds of all but one metric are better than the classification accuracy achieved by fine tuning thresholds in [146]. The threshold that does not give a good classification accuracy is the one for the export coupling metric ($AEC$) which is, again, consistent with the results reported in the literature.

Given the good classification accuracy above, logarithmic regression models can now be fitted on all size-threshold pairs from all systems/versions, which will be

Table 4.4 Classification accuracy of thresholds (t) estimated for the Ivy system versions.

| Coupling Metric | v1.1 | | v1.4 | | v2.0 | | Average |
|---|---|---|---|---|---|---|---|
| | t | g-mean | t | g-mean | t | g-mean | |
| IAC | 2 | 0.59 | 2 | 0.69 | 2 | 0.7 | 0.66 |
| ICC | 2 | 0.66 | 4 | 0.75 | 5 | 0.75 | 0.72 |
| IDC | 2 | 0.61 | 2 | 0.69 | 3 | 0.72 | 0.67 |
| IIC | 2 | 0.67 | 4 | 0.73 | 5 | 0.77 | 0.72 |
| AEC | 3 | 0.5 | 14 | 0.34 | 19 | 0.31 | 0.38 |

the models used to estimate the normalization thresholds for the coupling metrics of any DP participant class from any system given only its size as well as the slope and intercept of the metric's model (shown in Table 4.5).

To test these latter models, and also to compare their estimated thresholds with the thresholds that are calculated by the logistic regression based method (by using Eq. (4.19) before and after applying the correction), they are all used to classify classes in the largest system in the dataset (i.e. Eclipse JDT Core 3.4 - 994 classes). The obtained classification accuracies are shown in Table 4.6.

With regard to the logistic regression thresholds, it can be seen that thresholds calculated before applying the correction are larger than those calculated after. This is an expected consequence of the imbalance in the Eclipse dataset (only 20.72% of the classes are defected), which leads to risk underestimation [100] and, hence, the calculation of higher thresholds. Applying the correction has solved this problem

Table 4.5 Slopes and intercepts of the thresholds estimation models.

| Coupling Metric | Slope | Intercept |
|---|---|---|
| IAC | 1.08321 | -4.2026 |
| ICC | 2.16317 | -8.0504 |
| IDC | 2.58091 | -11.5766 |
| IIC | 2.19959 | -7.9253 |
| AEC | 13.53505 | -60.6759 |

Table 4.6 Classification accuracy of thresholds (t) estimated for Eclipse JDT Core 3.4

| Coupling Metric | Logistic Regression | | | | Logarithmic Regression | |
|---|---|---|---|---|---|---|
| | Before Correction | | After Correction | | After Correction | |
| | t | g-mean | t | g-mean | t | g-mean |
| IAC | 6 | 0.65 | 4 | 0.65 | 3 | 0.59 |
| ICC | 15 | 0.61 | 11 | 0.66 | 7 | 0.7 |
| IDC | 20 | 0.34 | 15 | 0.42 | 6 | 0.65 |
| IIC | 14 | 0.69 | 10 | 0.71 | 7 | 0.69 |
| AEC | 42 | 0.37 | 32 | 0.41 | 33 | 0.4 |

and improved the classification accuracy achieved with the thresholds of all metrics. Thresholds that are estimated by using the logarithmic regression models, on the other hand, are also better than those calculated before the correction in all but one case, and comparable with those calculated after.

It is worth noting the fact that while the logistic regression thresholds are calculated by fitting models on all classes of the Eclipse system, with information about their defect status, the logarithmic regression thresholds are calculated by fitting models on size-threshold pairs from all systems, in which the Eclipse system is represented by only one data point.

### 4.4.4 Using Thresholds for Normalisation

The values of the five coupling metrics of any DP participant class $c$ from any system can easily be normalized by using their corresponding thresholds, which are estimated based on the system size as well as the metrics' slopes and intercepts in Table 4.5, by using the following equation:

$$Metric^*(c) = \begin{cases} Metric(c) & if\ (slope \times \ln(size) + intercept) \leq 1 \\ \frac{Metric(c)}{slope \times \ln(size) + intercept} & Otherwise \end{cases} \tag{4.21}$$

where $\ln(size)$ is the natural logarithm of the software size.

The rationale of the equation above can be explained as follows. In systems that are small enough, as determined by the fitted models, no normalizations are required. For example, if a class is coupled to 20 other classes in a small 40 class system where the estimated threshold is less than one, the normalized metric value will still be 20. As the system size increases, the estimated threshold also increases, and the normalized metric value decreases. If the class above was in a 400 class system where the estimated threshold is 3, for example, the normalized metric value will be 6.67. This is the mechanism by which the increase in software size is taken into consideration when calculating the values of the coupling metrics.

## 4.5 Conclusions

This chapter has paved the way for filling the gap of evaluating the DP impact on software quality based on OO metrics by preparing a suitable set of quality metrics. Some of the metrics in this set are existing ones that needed only to be formally defined while the definitions of others have been slightly adapted. New coupling metrics have also been introduced and empirically validated as quality indicators. Moreover, new threshold calculation models have been developed and tested in this chapter, which can be used to calculate thresholds on the fly to normalize the coupling metrics based only on and relative to software size.

If classes playing a DP role are found to have a significantly different values in one or more of the quality metrics, these metrics will be included as input features in the role's recognition models (Chapter 6), and conclusions will be drawn with regard to the effect of including the quality metrics on the recognition accuracy.

# Chapter 5

# Dataset Construction

There is currently a lack of accurate and sufficiently large datasets of DP instances. This lack has given rise to the shortcoming of using subjectively labelled training datasets to train DP recognition models as well as the shortcoming of analysing the DP impact on software quality based on inaccurate and small datasets (as respectively discussed in Sections 2.2.10 and 2.3.3 in Chapter 2). The aim of this chapter is to remedy this lack by constructing large and reasonably accurate datasets for different DPs. To this end, existing attempts to construct DP datasets are briefly reviewed in the first section. Then, in Section 5.2, the challenges and obstacles to their construction, which have caused this lack in the first place, are discussed. A new approach to DP dataset construction is then proposed in Section 5.3, and the proposed approach is implemented in Section 5.4 to recover DP instances from 539 open source software systems. Finally, Section 5.5 discusses the outcomes of this chapter and how the work presented here relates to the rest of the thesis.

## 5.1 Existing DP Datasets

Although no standard or widely accepted benchmark datasets exists, several attempts have been made towards constructing one. The following lists and briefly discusses these attempts (ordered chronologically):

- Pattern-like Micro-Architecture Repository (P-MARt[1]) [73]: This dataset is the only peer-reviewed dataset of DP instances. Although the dataset's authors do not claim that all DP instances in the few studied systems have been recovered, they have expressed their confidence that most of them have based on the repetitive analyses performed by different teams [77].

- DEsign pattern Evaluation BEnchmark Environment (DEEBEE) [70]: This data set simply contains the results of automatic as well as manual analysis[2] of only five (C++ and Java) open source systems. It is freely available online in a website in which the results can be browsed and evaluated. This work is intended to be a first step towards a widely accepted benchmark.

- Design Pattern Benchmark (DPB) [14]: This dataset also contains the recognition results of analysing few open source systems by several tools. It is available online in a website in which DP instances can be searched, evaluated and voted for. The goal of the authors of this dataset is to construct a large dataset of community validated DP instances.

- PattErn Repository and Components Extracted from OpeN Source software (PERCERONS) [10]: This dataset is currently the largest available dataset with DP instances from 537 open source systems in its latest version [9], and it has been constructed by simply combining the results of two tools although the

[1] http://www.ptidej.net/tools/designpatterns.
[2] The manual analysis is performed in only one of the C++ systems.

tools' accuracy is admittedly low. Unlike the previous datasets, this one is initially intended to be a reuse repository.

- Software Engineering Research Center benchmark (SERC) [137]: This benchmark dataset was constructed to evaluate the accuracy of a DP recognition tool developed by the dataset's authors, and it was constructed by evaluating and merging the recognition results of other tools with that of their own tool. The inclusion of the recognition results of their own tool in the construction process may have biased the benchmark, one sign of this bias is the almost perfect precision/recall claimed for their tool.

None of these datasets of DP instances are prepared for the purpose of providing accurate and sufficiently large training datasets. The DEEBEE, DPB and SERC datasets have all been constructed by merging the results of existing DP recognition tools, and their ultimate aim is to construct an evaluation benchmark. The P-MARt repository is also aimed at providing an evaluation benchmark, but unlike the aforementioned ones, the peer-reviewed process of its construction makes its more accurate and reliable. All of these four benchmarks are small as they include DP instances from only few (less than 10) systems. The PERCERONS repository, on the other hand, does provide a sufficiently large dataset of DP instances, but it is probably highly inaccurate given the way in which it was constructed, as discussed above.

## 5.2 Challenges of Constructing DP Datasets

The ideal method of constructing a dataset of DP instances may be having a sufficiently large number of open source systems manually analysed by a team of experts in order to recover a comprehensive and complete list of peer reviewed DP instances from each system. In such a case, the recovered DP instances can be used

as positive training examples while all other permutations of classes can be used as negative ones. However, this method will be prohibitively time consuming as it requires careful study and deep understanding of each of the analysed systems, which may have obsolete or no documentation at all. Pettersson *et al.* suggest that using this method to construct a benchmark dataset is beyond the ability of a single research group [132].

The resource constraints are, however, not the only obstacle as it is sometimes almost impossible to objectively determine whether or not a set of classes form a true DP instance [17]. The boundaries of what does and does not constitutes a true DP instance may be "seldom or never completely clear-cut" [132] and, consequently, different evaluators and research groups may classify the same instances differently. This should be an expected consequence of the high degree of intrinsic subjectivity associated with labelling DP instances [173]. The fact that each DP has variants further complicates the problem of subjectivity as it raises the question of where to draw the line between what is a variant and what is not actually a DP instance.

It may be possible to reasonably overcome the obstacles above by focusing on and repetitively analysing a small number of systems, a good example of which is the peer-reviewed P-MARt dataset. It is, however, certainly not feasibly possible to do the same for a large number of systems in order to produce an accurate and representative dataset. The problem of having small datasets is apparently not limited to software DPs but, as noted in [101, page 729], "most software data sets are relatively small and new data are difficult to obtain".

## 5.3 ACODD: Automatic Construction of DP Datasets

The ACODD approach is proposed in this section in order to enable the construction of a large and reasonably accurate datasets. The approach is basically based on

analysing as many open source systems as possible by using as many DP recognition tools as possible, and then combining their results to produce a dataset of instances for each DP. The results are automatically combined by calculating how many votes each recognized instance receives (i.e. how many tools recognise an instance). DP instances that receive multiple votes are added to the dataset as positive examples while those with only one vote are added as negative examples.

The ACODD approach is based on the assumption that the more tools a DP instance is recognized by, the more likely it is to be a true positive instance. This assumption is substantiated by the following arguments. Since different recognition tools use different sets of design characteristics, recognition rules and recognition analysis techniques, DP instances that are recognised by multiple tools have a better chance to be true positive instances (i.e. instances with the right *intent*), and hence used as positive DP examples. On the other hand, DP instances that are recognized by only one out of several other tools are more likely to be false positive instances, and hence used as negative DP examples.

### 5.3.1 Negative and Positive DP Examples

It may be argued that, following the assumption above, instances with zero votes would be better negative examples than those with one vote. However, instances with one vote are preferred and used as negative examples for two reasons. First, instances with zero votes are simply all the permutations of classes that have not been recognized by any tool as instances of the DP in question, which will generate too many negative examples compared to the positive ones. In other words, using zero vote instances will result in huge and severely imbalanced datasets, which has a significant impact on the performance of many learning algorithms [86].

The second (and more important) reason is that one vote instances would probably be more characteristically similar to the positive examples than zero vote

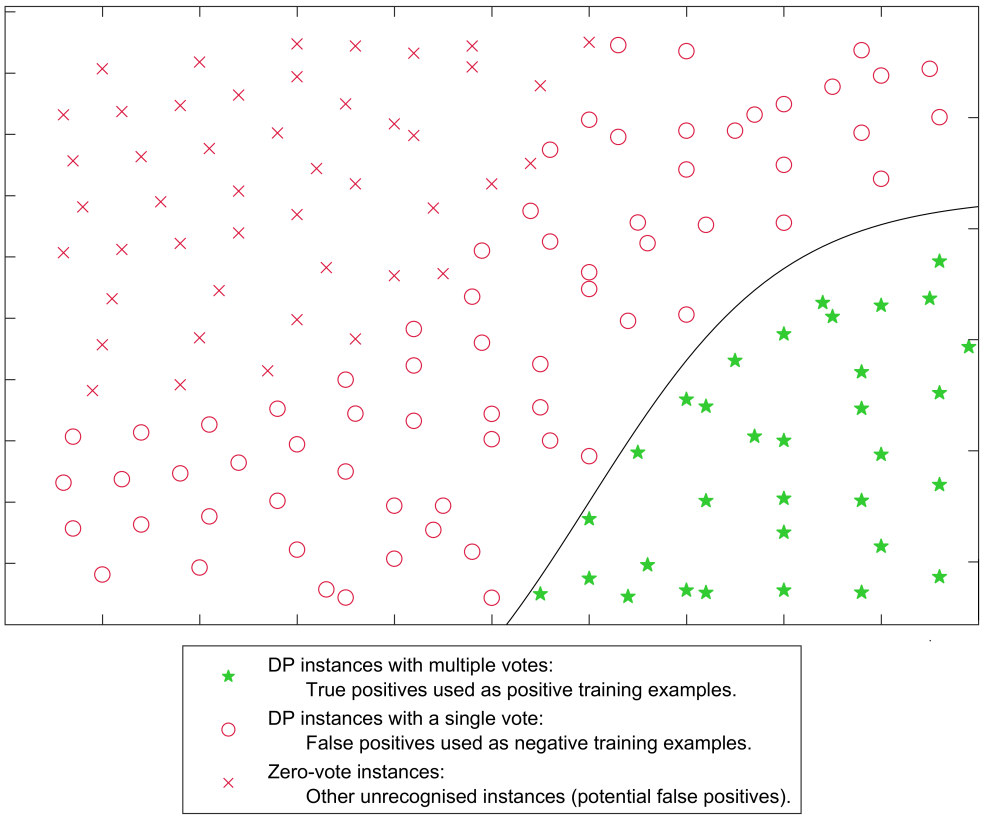



Fig. 5.1 An ideal example of how classification decision boundary should look like.

instances, and so, drawing the classification decision boundary between the positive examples and their closest negative examples may improve the recognition accuracy. This idea is visually illustrated in Fig. 5.1. Selecting negative examples based on their similarity to the positive ones has, in fact, been done before, though using a different method to select these similar examples. In [77], classes playing a DP role ($R_1$), which also have a similar structure to that expected for another DP role ($R_2$), have been added as negative examples to the dataset of the $R_2$ role.

The more DP recognition tools used, the more (1) accurate and (2) representative the constructed datasets will be. With regards to the accuracy, using more tools will enable increasing the minimum number of votes set for the positive examples, which are expected to improve their accuracy. The accuracy of the negative examples should also increase because, the more other tools by which negative examples are

not recognized, which can be seen as votes for rejection, the more likely it is that they would indeed be false positives. It cannot obviously be guaranteed that none positive examples will actually be false positives, nor it can be guaranteed that none negative examples will actually be true positives. However, such mislabelled training examples can be considered as *noisy* ones, which are normally found in most real-world datasets [35].

With regards to the representativeness of the constructed datasets, increasing the number of tools used will increase the range of DP variants that can potentially be added to the datasets. For example, when two tools are used and a minimum of two votes is set for the positive examples, all the positive examples in the constructed DP dataset will be limited to only one subset of DP variants that are recognizable by both tools. If, however, six tools are used and a minimum of three votes is set for the positive examples, the positive examples can potentially include instances for up to 20 (i.e. $\binom{6}{3}$) subsets of DP variants.

### 5.3.2 How ACODD Addresses the Identified Challenges

The ACODD approach provides a reasonable and feasible solution that addresses all the challenges discussed in the previous section. Automating the whole process of DP recognition and vote counting helps to avoid the resource constraints of the manual approach. Automation also helps to address the subjectivity problem although some level subjectivity may still be introduced into the datasets as a result of the subjectivity embedded in the recognition tools used to construct them. However, this problem is mitigated by relying on the consensus of multiple tools to identify the positive examples. Also, using one vote instances as negative examples can reverse the influence of any subjective and incorrect recognition rules in the tools that have recognized these one vote instances. So, the knowledge incorporated in the datasets will not simply reflect the knowledge embedded in the individual

DP recognition tools used to construct them, but will instead surpass them by building on their consensuses and learning from their mistakes.

### 5.3.3 ACODD Relationship to Existing Work

The ACODD approach has some similarities to previous work in the field. Pettersson *et al.* have suggested using what they call a *pooling process* in which the recognition results of different DP recognition tools are combined and verified, to gradually and semi-automatically construct a gold standard dataset [132]. It can be said that the DEEBEE and DPB benchmark datasets are intended to enable such a *pooling process* but with community voting used as the verification method. The ACODD approach can also be seen as an implementation of this *pooling process* but in a fully automated way that relies on tools voting. The main assumptions on which the ACODD approach is based, as explained in the introduction of this section, is similar to the assumption used in [102] to implement a data fusion recognition approach, in which the recognition results of different tools are combined to produce more accurate results. A similar idea is also used in [137] to evaluate the *trustworthiness* of DP instances.

## 5.4 Implementation of the ACODD Approach

The ACODD approach is implemented in this section to construct datasets for the DPs that are studied and experimented with in this thesis. The following subsections discuss the elements and steps required to construct the datasets.

### 5.4.1 Voters

Two criteria have been set for recognition tools to be used as voters. The first criterion is that they have to work on Java systems. This restriction is enforced to be

consistent with the systems in the only peer-reviewed dataset (i.e. P-MARt), which are going to be used as test systems. This consistency between the programming language of the systems from which the training dataset will be produced and the test systems is important due to the influence languages have on the distribution of metric values [176], and a number of examples for this influence are provided in [38]. The second criterion is that recognition results have to be exportable into, or easily convertible to, XML (i.e. Extensible Markup Language) format, which enables the recognition results to be automatically and easily manipulated.

Only eight of the available tools have met the two criteria, which are as follows:

- Design Motif Identification Multilayered Approach tool (DeMIMA[3]) [75].
- Design Pattern Miner tool (DP-Miner[4]) [53].
- FINe-grained DEtection Rules tool (FINDER[5]) [44].
- Metrics and Architecture Reconstruction PLugin for Eclipse (MARPLE[6]) [173].
- Pattern INference and recOvery Tool (PINOT[7]) [150].
- Rational Software Architect (RSA[8]).
- Similarity Scoring Algorithm tool (SSA[9]) [160].
- Web of Patterns tool (WOP[10]) [50].

The DP-Miner and RSA tools could not, however, be used as voters. The DP-Miner tool repeatedly fails and generates error messages. The error messages have been sent to its developers but no feedback has been received. For the RSA tool, the

[3]https://bitbucket.org/yann-gael/ptidej-5/.
[4]http://www.utdallas.edu/~jdong/DesignPattern/DP_Miner/index.htm.
[5]https://wiki.eecs.yorku.ca/project/dpd/.
[6]http://essere.disco.unimib.it/reverse/Marple.html.
[7]http://web.cs.ucdavis.edu/~shini/research/pinot/.
[8]http://www.ibm.com/developerworks/downloads/r/architect/index.html.
[9]http://users.encs.concordia.ca/~nikolaos/pattern_detection.html.
[10]http://www-ist.massey.ac.nz/wop/.

Table 5.1 DP recognition tools and the DPs they recognise.

| Tools | Adapter | Command | Composite | Decorator | F. Method | Observer | Singleton | State | Strategy | T. Method | Visitor | Proxy | A. Factory | Bridge | CoR | Facade | Flyweight | Mediator |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **DeMIMA** | R | R | R | R | R | R | R | R** | | R | R | R | R | R | R | R | R | R |
| **FINDER** | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | | R | R |
| **MARPLE** | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
| **PINOT** | R | | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
| **SSA** | R** | | R | R | R | R | R | R** | | R | R | R | | | | | | |
| **WOP** | R | | R | | | | R | | | R | R | R | R | R | | | | |

* R donates that the tool recognises the corresponding DP.
** Merged cells mean that the tool cannot distinguish between the DPs merged.

class names in the generated recognition results are not fully qualified class names, which means that the recognized DP instances cannot always be positively located.

### 5.4.2 DP and Roles

The DPs for which datasets are going to be constructed are a subset of the GoF DPs [71]. The selection of the GoF set is motivated by the fact that the test dataset (i.e. P-MARt) includes instances for only this set of DPs, and also by the fact that available tools almost exclusively recognize GoF DPs.

Table 5.1 shows the six recognition tools identified in the previous subsection and the DPs they recognise. Note that the DPs that Shi and Olsson suggest should not be targeted by recognition tools, based on their classification of DPs from a reverse-engineering perspective [150], are not included in the table for space limitation although some of the tools do claim to recognize them. One exception is the Command DP which is included to test the ability of the DP recognition approach proposed in this thesis to distinguish its instances from Adapter instances despite the fact that they are not distinguishable by one of the tools used to construct the datasets (i.e. SSA tool [160]).

Although datasets can be constructed for all of the DPs in Table 5.1, only a subset of them has been considered during the construction of the initial version of the ACODD datasets, which was reported in [3]. Including them all now would require re-analysing more than 400 systems included in the initial version, which are difficult and time consuming to do. The DPs for which datasets have been constructed are: Adapter[11], Command, Composite, Decorator, Observer, Visitor and Proxy. This subset, however, includes DPs from the structure-driven and the behaviour-driven categories in the Shi and Olsson's classification, which are the categories they suggest should be targeted by recognition tools.

During the recognition process, which will be explained in Chapter 6, only the key roles of each DPs are going to be considered. The rationale and justification of this decision will be included in the discussion of recognition process. What matters here is to note that the *key* roles are the ones deemed in [160] to be as such. Each of the seven DPs above has two key roles, and so, three separate datasets will be constructed for each DP: two datasets for the roles and one for the DP. The role datasets will be used to generate training datasets to train the classifiers that recognize candidate classes for each role in a search space reduction phase, and the DP datasets will be used to generate training datasets to train the classifiers that recognize DP instances in a subsequent phase. The two phases will be discussed in detail in Chapter 6.

### 5.4.3 Open Source Systems

A total of 539 open source Java systems have been analysed by using all of the six DP recognition tools in Table 5.1. The names of 430 systems have been taken from an earlier version of the PERCERONS dataset [10], which includes systems from a

[11]There are two versions of the Adapter DP [71]: Class-Adapter and Object-Adapter. The object version is the version used here because it is the one recognized by four of the six tools used while the other two (i.e. FINDER and MARPLE) recognise both versions.

wide spectrum of application domains (e.g. multimedia, games, communication as well as science and engineering). The other 109 systems are mainly Apache Software Foundation projects (e.g. Apache Jackrabbit[12]) and Java libraries (e.g. Java Net). The 539 systems differ in their sizes as well as their application domains. Such diversity is important as it leads to the generation of more representative training datasets.

In a study conducted on 988 open source Java systems to investigate the adoption of DPs by open source developers, it has been found that DPs are indeed used as an important implementation practice [80]. This gives a reassurance that at least most of the DP instances that are recognised by multiple tools will indeed be true positive instances as they are assumed to be.

### 5.4.4 Voting System

Instances of the seven DPs that are recovered from the 539 systems by using the six recognition tools are stored in XML based files. These files have different internal structures based on the tools that have generated them. So, to count the number of votes DP instances have received, a chain of programs have been developed to convert the XML files produced by each tool into a common format, and then to conduct the vote calculation. The vote calculation program places the instances in different folders based on the number of votes they have received. Different datasets can then be constructed from these folders by setting different minimum votes for the positive examples. The negative examples should always be the DP instances that have receieved only one vote, as discussed in the previous section. Fig. 5.2 shows the steps of the vote calculation process.

[12]https://jackrabbit.apache.org/

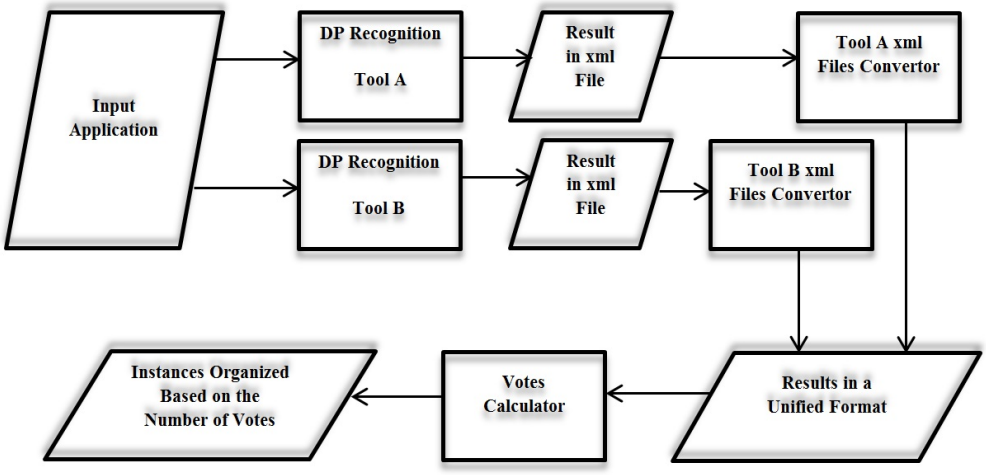



Fig. 5.2 Vote calculation process.

New voters can easily be added once they have produced their results in XML based files. All what is needed then is to convert them into the common format and re-run the vote calculation program.

### 5.4.5 Constructed Datasets

In order to construct the datasets, a minimum number of votes for positive examples needs to be set first. An appropriate minimum is the one that balances the number of instances at the selected minimum-vote level and the number of potential DP variant subsets that can potentially be represented in the constructed datasets as a result of the minimum selected. Let us assume, for the sake of the argument, that setting the minimum to 6, which is the highest possible minimum given the number of voters, will still produce a sufficiently large number of positive examples. Although this minimum will probably improve the accuracy of the constructed datasets, the range of the DP variants represented in the datasets will be limited to only one subset of variants that is recognizable by all of the 6 tools, which may affect the representativeness of the datasets. If, however, the minimum is set to 3, the constructed datasets may be relatively less accurate but more representative.

Table 5.2 Number of DP instances at different minimum-vote levels.

| Minimum Vote | **Adapter** | **Command** | **Composite** | **Decorator** | **Observer** | **Visitor** | **Proxy** |
|---|---|---|---|---|---|---|---|
| **1** | 1,238,893 | 97,296 | 9,331 | 7,011 | 18,086 | 3,505 | 1,384 |
| | (**305,811**) | (**66,171**) | (**3,471**) | (**2,799**) | (**9,392**) | (**1,691**) | (**768**) |
| **2** | 26,003 | 10,536 | 630 | 723 | 1,358 | 599 | **136** |
| **3** | 3,458 | 650 | **93** | **110** | **84** | **416** | 10 |
| **4** | **681** | **71** | 20 | 14 | 0 | 238 | 0 |
| **5** | 95 | 0 | 0 | 2 | 0 | 0 | 0 |
| **6** | 3 | 0 | 0 | 0 | 0 | 0 | 0 |

∗Between brackets bold numbers are the number of negative training instances and the other bold numbers are the positive ones.

Table 5.2 shows the number of DP instances at different minimum-vote levels. It can be seen that selecting 3 as a minimum is probably the best choice for the majority of DPs. For the Adapter and the Command DPs, however, a minimum of 4 may be more appropriate given that they are not distinguishable by the SSA tool, as mentioned earlier, and increasing the minimum for these two DPs should compensate for the resulting inaccuracy. Although all the tools should recognize the Proxy DP, only three of them actually have recognized instances for it[13], which explains the relatively low number of instances for this DP at all minimum-vote levels, hence it is probably necessary to decrease its minimum to 2.

The row of the minimum-vote level of 1 in Table 5.2 shows two different numbers of instances for each DP. The number at the top represents the total number of instances that have been recognized by any tool in any system. The number between brackets below is the number of negative examples, which is the number of instances that have received one and only one vote in systems that have been successfully analysed by all the tools that recognize the corresponding DP. Some tools fail to successfully complete analysing some systems, and in such cases, not voting for the systems' instances does not necessarily mean a vote for rejection. So,

[13]The DeMIMA and the MARPLE tools have not recognized any Proxy DP instance at all. Although the PINOT tool has recognized Proxy DP instances, the names of the classes playing the RealSubject role are missing in all of the reported results.

not adding the instances that have been recognized in such systems by a single other tool to the set of negative examples is expected to improve the accuracy of the constructed datasets.

The difference between the number of instances that have received one vote and those with multiple votes shows the extent of disparity between the results of different recognition tools, which confirms the observations reported in the literature, as discussed in Chapter 2 (Section 2.2.10). The table that shows the number of role-playing classes at different minimum-vote levels for DP roles can be found in Appendix B.

## 5.5 Conclusion

A new feasible approach has been proposed in this chapter to enable the construction of large and reasonably accurate DP datasets. The fact that the proposed approach is a fully automated one facilitates the construction of large DP datasets, which would otherwise be an unattainable task. The approach is designed in a way that improves the accuracy of the positive as well as the negative DP examples. Since more recognition tools can be easily added as voters, the datasets constructed in this chapter can be continuously evolved towards larger and more accurate DP datasets. The set of datasets constructed in this chapter, as it stands now with 539 systems and six DP recognition tools, is by far the largest DP dataset currently available. All the constructed datasets are going to be made available online on a website[14] developed specifically for this purpose as well as on ResearchGate.

The constructed DP/DP-role datasets can be transformed into training datasets by calculating a set of features (e.g. the set introduced in Chapter 3) for each positive/negative example. The sets of features that will be used to represent the

[14]http://dpdatasets.com.

examples in the DP/DP-role training datasets, as well as the format of their training instances, will be discussed in Chapter 6.

# Chapter 6

# DP Recognition System

This is the chapter in which the contributions of all the previous chapters come together and put in use to build a DP recognition system. Building the recognition system is not, however, the aim itself but only a means. The aim is to evaluate and demonstrate the adequacy of the feature set introduced in Chapter 3 and the dataset constructed in Chapter 5 for the purpose of training DP classification models. The impact of DP on software quality, as measured by the quality metrics introduced in Chapter 4, will also be evaluated in this chapter. The quality metrics that capture a significant quality impact of a DP role will be added to the input feature vector of the classifier trained to recognise the role, and conclusions will be drawn with regard to the effect of adding the quality features on the recognition accuracy.

The chapter starts, in the first section, by introducing the overall design of the recognition system, which includes discussions of the two-phase approach adopted for the system as well as the structures of the training instances and the global feature sets designed and dedicated for the classifiers to be trained in each of the two phases. Section 6.2, then, discusses the subsets of features that are selected for DPs and DP roles and the feature selection algorithm used, as well as the impact of DPs on software quality. The process of building the classification models

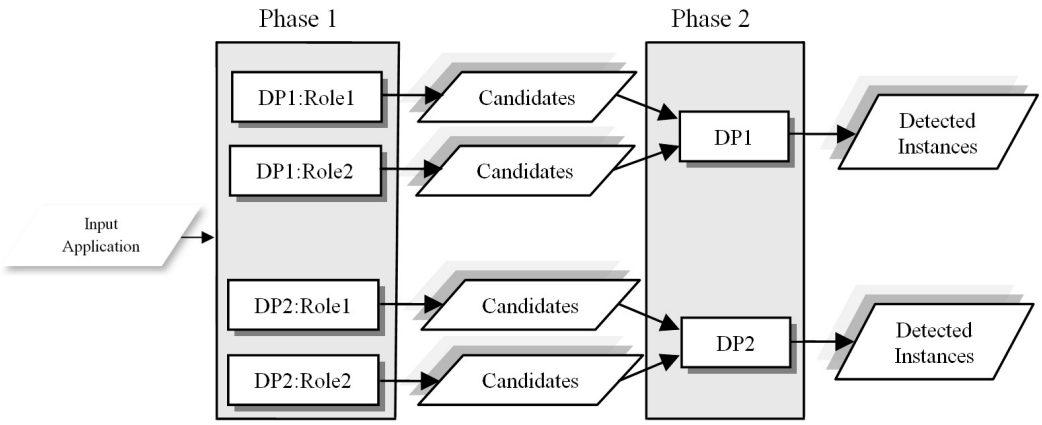



Fig. 6.1 The design of the recognition system

and the machine learning method used are introduced in Section 6.3. In Section 6.4, the results of the experiment conducted to test the classifiers trained in the preceding section are presented and discussed. The outcomes of this chapter are finally discussed in Section 6.5.

## 6.1 Recognition System Design

The overall design of the recognition system follows a two-phase approach. The first phase identifies a set of candidate classes for each DP role and, in the second phase, all combinations of the candidates of related roles are checked to recognize the ones that do constitute an instance of a DP (Fig. 6.1). The identification of candidate classes as well as the recognition of DP instances are performed using trained classifiers.

The following three subsections discuss, respectively, the benefits of adopting this two-phase approach and the DP roles considered in the first phase as well as the structures of training instances designed for each phase. The sets of features that form the global feature sets of the two phases, from which a feature subset is to be selected for each role/DP classifier, are then discussed in Section 6.1.4.

### 6.1.1 Two-Phase System

Breaking down the recognition process into two phases has two main expected benefits. First, reducing the search space by identifying a subset of a system's classes helps to mitigate the scalability problem, which is one of the main problems to be addressed as discussed in Chapter 2. If, for example, the instances of a DP that consists of four roles need to be recognized in a system that has 100 classes, without a search space reduction phase there would be $94,109,400$ (i.e. $\frac{100!}{(100-4)!}$) permutations to check. However, with a search space reduction phase that identifies as many as half the classes as candidates for each of the four roles, there will only be $6,250,000$ (i.e. $50^4$) combinations to check, which represents only 6.64% of the size of the original search space. In other words, the search space will be reduced in this case by 93.36%.

The second expected benefit of performing the recognition process in two phases is to improve the recognition accuracy by filtering out classes, in the first phase, that are certainly not playing any role in the DP to be recognized, which eliminates the possibility of falsely recognizing them as parts of false positive (FP) instances in the second phase. Also, performing the recognition process by using multiple and relatively less complex classifiers, instead of using one complex classifier, may be a source of accuracy improvement. It is in a way an implementation of the divide and conquer strategy in which the recognition problem is split into multiple smaller problems, each of which is solved by a classifier that uses input features relevant only to the problem at hand.

Given the purpose of the first phase as a search space reduction phase, its recall (i.e. the fraction of existing role-playing classes that are identified as role-candidates) is more important than its precision (i.e. the fraction of identified candidates that are not role-playing classes). This is because any mistakenly filtered out classes will decrease the recall of the overall recognition system while, on the other hand, any

FP candidate classes identified in the first phase can be discarded in the following phase. However, if too many FP candidate classes are identified, the benefits expected from having a search space reduction phase may be compromised.

### 6.1.2 DP Roles

Different DP recognition tools consider and recognize different sets of DP roles. The set of the roles[1] recognized is called the *occurrence type* and the size of this set is called the *occurrence size* [132]. The recognition system in this thesis adopts the same occurrence types as the ones used in [160] for each of the seven DPs included in the experiments reported in this thesis. The occurrence type adopted for each DP includes only two of their *key* roles. The *key* roles are the ones with distinctive and relatively less common characteristics. The Leaf role in the Composite DP, for example, have only one (i.e. generalization) relationship with other roles in the DP while, on the other hand, the Component and Composite roles are associated by an association and generalization relationships as well as iterative method invocations, which makes them the key roles to be targeted by recognition system.

The exclusion of non-key roles has several advantages for the training and testing (as well as the practical application) of the classification models and, consequently, the performance of the overall recognition system. The advantages are discussed as follows:

- **More training examples**: The larger the occurrence size, the less likely that DP recognition tools will agree on their recognized DP instances, and consequently, less positive training examples will be available for the classifiers in the second phase. In a ten class software system, and assuming that classes have a uniform probability to be recognized as role-playing classes within DP instances, there is $7.811^{-12}$ chance that a set of three DP recognition tools

[1] DP *roles* as well as other DP-related terminologies are defined in Section 2.1.2.

would agree on a particular instance of a DP if the occurrence size is four[2]. If, however, the occurrence size is reduced to two, the probability will be increased to $1.372^{-6}$.

- **More accurate datasets**: Since the SSA tool [160] is one of the tools used to construct the datasets, and since its occurrence type includes only two roles in all of the DPs included in this thesis, a decision has been made to include the same set of roles as those in its occurrence type. There are, however, two other options to deal with this case. The first option is to remove this tool from the set of tools used to construct the datasets, which will lead to reducing the minimum number of votes required for the positive training examples, rendering them less accurate. The second option is to count its partial agreement with DP instances recognized by other tools as a vote for the full instances, which will also result in degrading the accuracy of the constructed datasets. If, for example, the DP instances (`ClassA`, `ClassB`, `ClassC`) and (`ClassA`, `ClassB`, `ClassD`) are recognized by two tools, both instances will be added as positive training examples if the SSA tool recognizes (`ClassA`, `ClassB`) as an instance of the same DP. However, the classes `C` and `D` are of a less certainty that they actually play the roles they are assumed to play, which means that many of the tools' FPs may be injected to the datasets as positive training examples. Nevertheless, even if it can be guaranteed that they do represent true role-class mappings, the information added to the datasets by including all the (key and non-key) roles is likely to be less/not useful and noisy, as explained in the next point.

[2]Given that there are 4 roles each of which can potentially be mapped to/played by one of the 10 classes, the total number of the possible permutations is 5040. So, the probability of recognizing a particular roles mapping (i.e. DP instance) by one tool is $1.984^{-4}$, which is one over the total number of the possible permutations. The probability for the same instance to be recognized by three tools is then $(1.984^{-4})^3$ which equals to $7.811^{-12}$.

- **Less noise[3] in training data and feature sets**: The fact that non-key roles have less distinctive characteristics means that most of the variations in the characteristics of classes playing such roles are likely to be related to the randomness of real world systems more than they are related to the underlying models of DPs. So, the inclusion of non-key roles will add potentially irrelevant or noisy features to the global feature set of the second phase (the global feature sets are discussed below in Section 6.1.4). The first phase will, however, be more affected by their inclusion as it will include classifiers that are specifically trained for these non-key roles by using their potentially noisy training examples. Such classifiers will not be capable of identifying candidate classes for the non-key roles in a way that effectively reduce the search space while maintaining a perfect recall. This problem with the non-key roles has in fact been observed and reported in [78], in which a search space reduction approach is proposed (discussed in Chapter 2).

- **Single classifier for many variants**: Some of the non-key roles can actually be omitted (i.e. played by no class at all) in some DP instances, and a number of examples for such instances are given in [52]. So, such DP instances with one or more missing roles represent variants that need to be recognized. In order to recognize such variants, Balanyi *et al.*, for example, suggested using simplified DP recognition rules that do not include roles which can be omitted [17]. In the context of this thesis, one way to recognize such variants is to train a classifier for each variant, which is the option adopted by [16] as discussed in Chapter 2. Alternatively, one classifier can be trained for all variants if only the key roles are considered as they cannot be omitted in any instantiation of the DPs, which is the option adopted in this thesis.

[3]Noise is defined in [57] as "any property of the sensed pattern which is not due to the true underlying model but instead to randomness in the world or the sensors".

- **Improving system scalability**: Besides the scalability improvement achieved by having a two phase system, the exclusion of non-key roles can improve the scalability even further since the number of the combinations to be checked grows exponentially with the number of roles. As shown in the example given in the previous subsection, breaking down the recognition process into two phases reduced the search space from $94,109,400$ to $6,250,000$ (i.e. $50^4$) combinations to be checked. Using two key roles instead of four roles will reduce the search space further to only 2500 (i.e. $50^2$) combinations to be checked. If, however, all roles are included and several classifiers are trained for variants with missing roles, the combinations to be checked will increase to more than $50^4$ in this example.

One disadvantage for excluding the non-key roles is that information about the classes playing such roles (if any) will not be provided with the result of the recognition process. However, such information can easily be recovered manually as they are usually closely related to classes playing the key roles [160]. The decision to exclude non-key roles is not limited to DP recognition systems, but also include some studies on the quality impact of DPs (e.g. [99]).

### 6.1.3 Training Instances

Some DP roles can be played by/mapped to multiple classes in single instantiations of DPs. The ConcreteCommand and the Receiver roles in the Command DP, for example, are both expected to be played by multiple classes, in which each class playing the ConcreteCommand role encapsulates a request in terms of a set of operations to be performed on its corresponding Receiver class. Each possible combination of role mappings can be counted as a separate instance or, alternatively, they can be merged into a single instance based on their shared *anchor* role-playing classes (e.g. the class playing the Command role in the previous example). Dif-

ferent DP recognition approaches use different counting methods [169]. While, for example, related DP instances are merged in [115], they are considered to be separate ones in [44].

The decision as to whether to merge the DP instances or to treat them as separate ones determines how DP instances (in the datasets constructed in Chapter 5) are transformed into and represented as training instances, which can consequently have an impact on the performance of the trained models. The following subsection discusses the problems of the current transformation and representation approaches, and then the next subsection introduces a new approach to be used in this thesis.

#### 6.1.3.1 Problem of DP Instance Representation

In order to train the machine learning classifiers that implement the DP recognition system, the datasets constructed in Chapter 5 need to be transformed into a format that can be used as an input to the training process. In this format, each training example, or training instance, should represent an individual example of the concept (i.e. DP/DP role) to be learned, and each instance should be characterized by an input feature vector along with the instance's target label. In other words, the DP/DP role instances need to be transformed into training instances.

The datasets of the DP roles, an excerpt of which is shown in Table 6.1, contain lists of individual classes each of which either plays the role (i.e. positive

Table 6.1 An excerpt from the Adapter role dataset.

| Class Name | Label |
|---|---|
| `com.globalretailtech.data.Employee` | NotAdapter |
| `org.apache.jackrabbit.rmi.server.ServerObject` | NotAdapter |
| `de.oio.jpdfunit.document.pdflibimpl.PdfBoxContentAdapter` | Adapter |
| `moa.classifiers.trees.AdaHoeffdingOptionTree` | NotAdapter |
| `de.quippy.mp3.decoder.LayerIDecoder` | Adapter |
| `org.apache.ctakes.core.fsm.token.adapter.WordTokenAdapter` | Adapter |

example) or does not play the role (i.e. negative example). So, they can be easily transformed into training instances by replacing the class names with input feature values that are calculated for their corresponding classes. The target labels (e.g. Adapter/NotAdapter in Table 6.1) can also be simply replaced by +1/-1 in the training instances.

The problem, however, arises with the transformation of DP instances because of the multi role-playing classes issue discussed above. Also, the question of how to represent the relationships between classes playing different roles within DP instances is another issue to be addressed. Different previous recognition systems, in which training instances have been used, have used different approaches to deal with these two issues.

In [173] (discussed in Section 2.2.3), all the individual role-mappings recognized for a DP by applying the recognition rules are clustered into $k$ clusters. Then, the related mappings are merged together into a single instance represented as a set of $k$ Boolean features, which represent whether or not the merged instance includes a mapping in the corresponding cluster. This representation approach, with such level of information compression, provides the classifiers with very little information about DP instances. It can also degrade the classifiers' performance by inducing another source of variations between training instances that does not depend only on the different ways in which DPs can be implemented, but also on how many role-mappings are merged into different training instances[4]. In fact, the authors themselves have noted that the DPs in which different role mappings are always transformed into different training instances, and hence no clustering was used for them, have performed better than other DPs.

[4]For example, while a DP instance with a single role-mapping will be transformed into a training instance with a feature vector of zeroes and a single one, the training instance of another DP instance that happens to have several role-mappings may have as many ones as there are mappings.

In [39] (discussed in Section 2.2.5), different value aggregation methods (e.g. mean) are used to aggregate the values calculated for the features of any set of multi role-playing classes. Such aggregated values may not, however, be a good representation for the individual values of the aggregated classes, especially given the fact that the features used are all OO metrics that mostly have skewed distributions [113] [93]. So, the classifiers that are trained by using aggregated training instances may perform poorly when used to recognize DP instances in a test system. Another problem with this representation approach is that the relationships of classes within DP instances are ignored and not represented in the training instances, as discussed earlier.

In [51] (discussed in Section 2.2.9), DP instances are assumed to not be representable as regular training instances, but rather as sets of inter-connected training records in which each record represents a class. Although such assumption is clearly not correct, it shows, along with the approaches discussed above, that the problem of transforming DP instances into training instances has not yet been solved in an effective and suitable way.

#### 6.1.3.2 DP Instance Representation

In order to properly transform DP instances into training instances, the two issues identified in the previous subsection need to be addressed in a way that avoids the shortcomings of the previous approaches. As to the first issue (i.e. multi role-playing classes), it can simply be solved by transforming each individual role-mapping in DP instances into a separate training instance. This solution has the following three advantages.

- It leads to creating more DP training instances that provide a richer representation for a wider range of possible role-mappings.

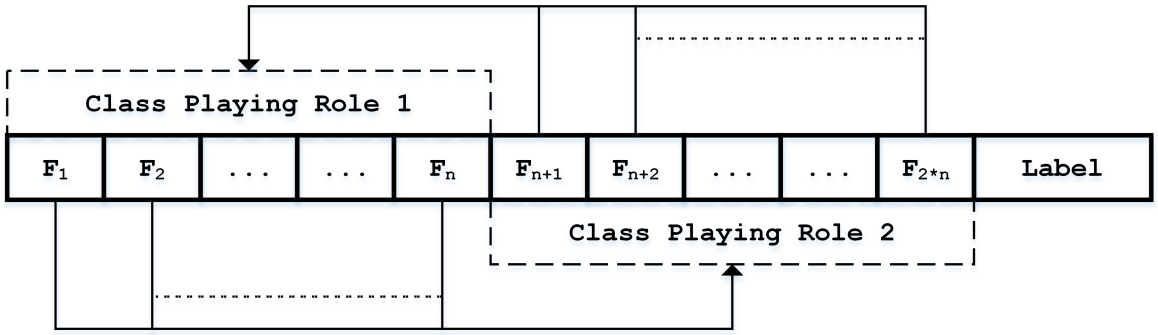



Fig. 6.2 Input feature vector for two-role DP training instances with $n$ relationships.

- It preserves and accurately represents the characteristics of individual role-mappings instead of merging them in a way that may obscure or distort them.

- Since DP instances exist in software systems (if any) as individual role mappings, it is intuitively better to use training instances that also represent individual role mappings.

The numbers reported in Table 5.2 represent the number of non-merged DP instances in the constructed datasets, which also now represent the number of training instances to be created.

The issue of representing the relationships between classes within DP instances can be solved by having a feature vector in which each subset of features represents the outgoing relationships (e.g. association or method invocation) from one role-playing class to another role-playing class. So, if there are, for example, three roles and ten relationships, the feature vector will be 60 feature long. Since DPs have different number of roles, their training instances can be represented by feature vectors of different lengths. However, since only two roles are considered for each DP in this thesis, they will all be represented by feature vectors of the same length.

Fig. 6.2 visually illustrates how the structure to be used for DP training instances looks like. If, for example, the first slot in the part of the first role-playing

Table 6.2 An excerpt from the Composite DP dataset.

| Class Playing Role 1 | Class Playing Role 2 | Label |
|---|---|---|
| `org.da.expressionj.model.Expression` | `org.da.expressionj.expr.CodeBlock` | Composite |
| `org.da.expressionj.model.Expression` | `org.da.expressionj.expr.ExprArray` | Composite |
| `org.apache.bcel.classfile.Attribute` | `org.apache.bcel.generic.ClassGen` | NotComposite |
| `org.jgap.Gene` | `org.jgap.impl.CompositeGene` | Composite |
| `cartago.manual.syntax.Term` | `cartago.manual.syntax.Structure` | Composite |
| `org.apache.any23.http.HTTPClient` | `org.apache.any23.Any23` | NotComposite |

class (i.e. $F_1$ feature) represents a certain association feature, the first slot in the part of the second role-playing class (i.e. $F_{n+1}$ feature) should also represent the same association feature. So, as can be seen in the figure, while the $F_1$ feature represents the association from the first class to the second, the opposite direction is represented by the $F_{n+1}$ feature.

Having addressed the two issues and designed the feature vector of DP training instances, the DP datasets constructed in Chapter 5, an excerpt of which is shown in Table 6.2, can now be transformed into training instances by replacing the name of the two role-playing classes with their inter-relationship features, as shown in Fig. 6.2. The target labels can also be simply replaced by +1/-1 in the training instances.

### 6.1.4 Global Feature Sets

Although the structure of the feature vectors for both DPs and DP roles have been designed, the exact features to fill in the slots in the feature vectors have not yet been specified. In this section, two sets of features are going to be selected from the structural and behavioural features that have been introduced in Chapter 3 and used to fill in the slots of the feature vectors of DPs and DP roles. The two sets will then be considered to be the global feature sets from which different subsets are to be selected for the classifiers in phase one (i.e. DP roles) and phase two (i.e. DPs).

#### 6.1.4.1 Global Feature Set in Phase One

Since that the aim of the classifiers in phase one is to identify a set of candidate classes for each DP role, and the fact that they are going to be trained by datasets of individual classes, it can be said that they are basically working on the intra-class level. So, they need to be fed with information on this same level, which is exactly what the 23 intra-class features (introduced in Section 3.2.1 in Chapter 3) provides. Nevertheless, the types of incoming and outgoing inter-class relationships from and to other classes are also important at the class level. For example, a class without any outgoing association and method invocation of any type will definitely not be an Adapter class, and should therefore be filtered out as a non candidate. As discussed in Chapter 3, there are structural as well as behavioural inter-class relationship features, and the following discusses how they are going to be incorporated and represented at the intra-class level.

For the structural relationships, namely the generalization and association relationships, each relationship will be represented by two binary features, one for each of the incoming and outgoing directions. For example, the intra-class outgoing counterpart of the Association Through Array Attribute relationship feature, which is defined by the predicate $ATAA(c_1, c_2)$ for any two classes $c_1, c_2 \in C$, can simply be defined as follows:

$$ATAA_{in}(c_1) = \begin{cases} 1 & if\ \exists c_2 \in C \mid ATAA(c_1, c_2) \\ \\ 0 & Otherwise \end{cases} \tag{6.1}$$

and the incoming counterpart can be defined as follows:

$$ATAA_{out}(c_1) = \begin{cases} 1 & if\ \exists c_2 \in C \mid ATAA(c_2, c_1) \\ \\ 0 & Otherwise \end{cases} \tag{6.2}$$

The definitions of the intra-class counterparts of other inter-class structural relationship features follow the same template, and they only require substituting the $ATAA()$ predicate with the appropriate predicates. Adding these features to the global feature set in this phase brings its total number of features up to 43 features.

Representing the inter-class behavioural relationships at the intra-class level is not, however, as straightforward as it was the case with the structural ones. This is because they are defined to count the number of method invocations between classes for different invocation types. If they are, however, binarized by following the same approach as used for the structural ones above, they would convey less information which can better be represented by using much fewer features. So, instead of using all of the 128 features (i.e. 64 for each of the incoming and outgoing directions) which, as explained in Section 3.3.3, represent all possible combinations of the four method invocation characteristics for each of the four invoker-invokee interrelationship levels, only 16 binary features can be used to represent each individual invocation characteristic and interrelationship level in both incoming and outgoing directions.

Two more features will, however, need to be added to 16 features above in order to capture the existence of regular (i.e. not delegation) method invocations in both directions. This is because the absence of a delegation does not imply the absence of a regular invocation, and such information will be missed if only two (incoming and outgoing) binary features are created for the similarity characteristic of method

invocations, as they will only capture the existence of method delegations. In fact, this method invocation characteristic represents two categories (i.e. delegation versus regular method invocations), and such categorical features with two possible values should normally be converted into two *dummy* features [96].

The total number of features added to the global feature set in phase one is 61 features, which convey just the information required in phase one. In the dataset versions of the absolute feature values, the two denominator features (i.e. $M_{DEC}(c)$ and $A_{DEC}(c)$) are taken out of the global set as they are already included in the calculation of the values of other features, making the total number of features in these datasets 59 features.

#### 6.1.4.2 Global Feature Set in Phase Two

The classifiers to be used in phase two are, as mentioned earlier, expected to recognize which combinations of candidate classes do constitute DP instances, and so they need information about the type, the number and the direction of the relationship connections between the candidate classes. The structural as well as behavioural inter-class relationship features, as they are defined in Chapter 3, do provide this information for any pair of candidate classes. There are 74 such features that will create a global feature vector with 148 features which will capture the relationships in two directions as shown in Fig. 6.2. The intra-class features have not been included here because they have already been used in the previous phase, and also because the information they hold is probably not relevant to the task assigned to this phase.

## 6.2 DP Features and Quality Impact

Having specified the structure of phase one's and phase two's training instances, in which all features from their global feature sets are represented, and having transformed the datasets constructed in Chapter 5 into training datasets with the specified structure, different feature subset can now be selected for each individual classifier to be trained. The benefits and importance of using feature subsets, as opposed to using the whole feature sets, is discussed in the following subsection, which also introduces the algorithm used for this purpose. Then, in Section 6.2.2, the impact of DPs on the quality of their participant classes are tested (role by role) based on the quality metrics introduced in Chapter 4. The metrics that capture a statistically significant impact will be used to extend the feature subsets that are selected for the corresponding DP role.

### 6.2.1 Feature Selection

Feature selection is the process of identifying a subset of features that are informative and relevant to the target concept. The relevance of a feature or a feature subset in classification problems corresponds to its discrimination ability [159] [166]. The selected subset should then provide a representation in which instances of the same role/DP are close to one another and far from those that are not. Such a representation is also the one in which "the true (unknown) model of the patterns can be expressed" [57, page 7].

Identifying relevant features is a critical and crucial step towards successfully solving any pattern classification problem [130]. This is because that, with the presence of a large number of irrelevant and noisy features, the signal-to-noise ratio is decreased in the training datasets, which may greatly degrade the accuracy of the trained classification models [96]. Model training in such situations also carries

the risk of over-fitting to aspects that are irrelevant to the target concept [32]. The advantages of removing irrelevant features, besides improving the generalization performance of the trained classifiers, include that it results in less training data which reduces the training time and that it helps to gain a better insight into the target concept [47].

#### 6.2.1.1 Feature Selection Method

The fact that features which are relevant to different roles/DPs are not known a priori (at least not all of them), which is often the case in real-world problems, means that feature selection methods are indispensable to the objective identification of such features. There are two main categories of feature selection methods: subset selection and feature ranking [96]. In subset selection methods, the global feature set is (heuristically) searched for the optimal subset. Two methods from this category have been experimented with (i.e. consistency [112] and correlation [82] based subset evaluators, as implemented by WEKA[5] [81]). However, they have both yielded too small (less than 5 feature) subsets for some DPs, which have led to poor classification accuracy, as would be intuitively expected.

The second category (i.e. feature ranking) is a widely used category of selection methods in which individual features are evaluated independently and scored according to a given criterion that quantifies the utility of the features. Then, all features that have a score lower than a certain threshold are discarded. Although ranking methods are simple, scalable and have had empirical success, they are criticised for the possibility of selecting a subset with redundant features. However, adding presumably redundant features to the selected subset may still be beneficial as even high feature correlation does not necessarily imply the absence of feature complementarity. If, on the other hand, there exist complementary (interacting)

[5]Waikato Environment for Knowledge Analysis.

features that are individually less- or irrelevant, such features may be assigned a low score and discarded, which is a shortcoming of this category of methods [79].

The selection method used in this thesis is based on using an information theoretic measure (the mutual information, in particular) as a criterion to evaluate the relevance of individual features. A fundamental unit of information, based on which the mutual information criterion is calculated, is the *entropy* which quantifies the uncertainty present in the distribution of a variable. The entropy of the class label ($Y$) distribution is expressed in terms of the prior probabilities of class label values as follows [159]:

$$H(Y) = -\sum_{y \in Y} P(y) \log_2(P(y)) \tag{6.3}$$

where $P(y)$ is estimated based on the relative frequency of a class in the dataset. The class label variable $Y$, which has two values (i.e. $+1/-1$) in our case, receives its maximum entropy when all of its values have equal probability. The uncertainty of the class identity after knowing the value of a feature $X$ can be calculated by using the conditional entropy as follows:

$$H(Y|X) = -\sum_{x \in X} P(x) \sum_{y \in Y} P(y|x) \log_2(P(y|x)) \tag{6.4}$$

Where $P(x)$ is the fraction of observations having the value $x$. The mutual information between $Y$ and $X$ is defined as the reduction in the uncertainty of the class identity after knowing the value of feature $X$, which can be calculated as follows [159]:

$$I(Y, X) = H(Y) - H(Y|X) \tag{6.5}$$

The value of $I(Y, X)$, as a relevance score, is calculated[6] for each feature in the datasets of every DP and DP role. It is worth noting that, in the experiments conducted during the development of the recognition approach proposed in this thesis, other measures have been used to calculate the relevance scores, including Fisher score [57] and ReliefF [103], but the mutual information measure is the one that has consistently led to good classification performance.

#### 6.2.1.2 Feature Subsets

Having scored and ranked the features, a cutting criterion (i.e. a threshold for feature acceptance) needs to be determined in order to select the final feature subsets that will comprise the input of the classification models to be trained. There are several possible cutting criteria for this purpose [12], and the one used is that based on setting a threshold over the relevance scores. Since there are a total of 42 training datasets for the 7 DPs and 14 DP roles (i.e. 21 for each of the two feature calculation methods introduced in Chapter 3) in which the scale and distribution of feature scores vary, a single universal threshold may not work suitably for all of them.

To solve this problem, the median (or the second quartile) of above-zero scores in each dataset is used as the threshold, and features with lower scores are discarded and not used in the input of the corresponding classification models. However, in five DP datasets, less than 15% of the total number of features have been selected by using this threshold, in which case the threshold used is lowered to the first quartile in order to allow for the inclusion of more features. The number of features selected as a result of lowering the threshold were still less than those selected in other DP datasets.

[6]The calculation is performed by using MIToolbox [32], which can be downloaded from http://www.cs.man.ac.uk/~pococka4/MIToolbox.html

For DP roles, the subsets of features selected based on the normalized and unnormalised versions of their datasets have about half the number of features in phase one's global set (i.e. $\approx 30$ features). For DPs, on the other hand, the subsets of features selected based on the unnormalised datasets have, on average, about 83% more features than those selected based on the normalized datasets, which shows that many features have lost their (potentially false) relevance due to feature normalization. The number of features in the selected subsets, in both (normalized and unnormalized) cases, varies from 24 to 74 out of 148 features in phase two's global set.

Most, if not all, of the features that are intuitively expected to be relevant have actually been selected. This is an interesting observation for two reasons. First, it shows that the negative training examples differ from the positive ones in exactly the features they would be different in if they are indeed false and true positives, respectively (as assumed in Chapter 5 where the datasets have been constructed). The second reason is that it provides an objective evidence supporting the intuitive assumption about relevant features. Contrary to the expectations, however, these intuitively expected features have received higher ranks in the unnormalised DP datasets (in the top 10, in most cases) than in the normalized datasets. The normalized Proxy DP dataset is the only one in which the expected features have not been selected, although other features capturing similar design properties have been selected. A detailed study of how the selected features match/mismatch what may be expected based on the description of DPs is out of the scope of this thesis. It is nonetheless an interesting research area that is left for future work.

The box plots in Fig. 6.3 show the difference between the positive and negative training instances in the distribution of UPM feature in the Adapter role normalized and unnormalised datasets as well as AIADU feature in the Composite DP

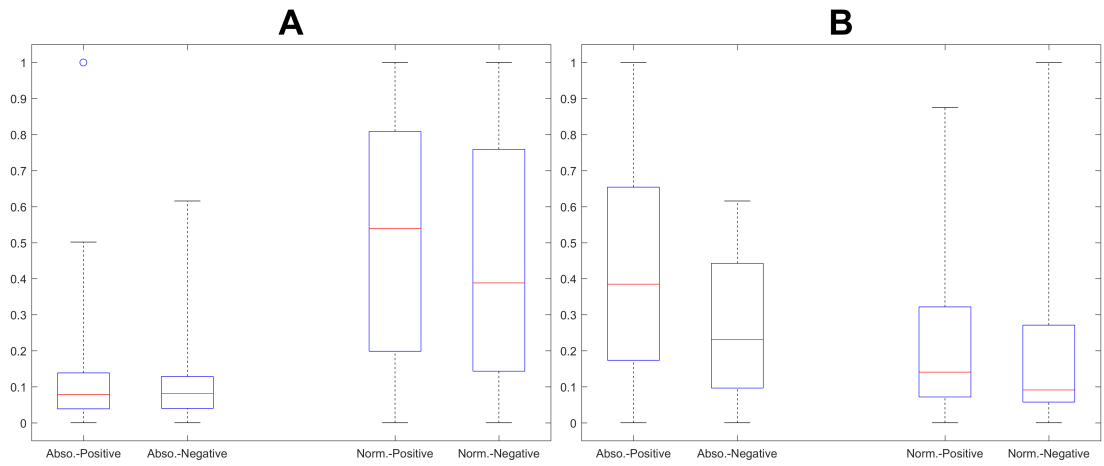



Fig. 6.3 Boxplots for two features in Adapter role (**A**) and Composite DP (**B**) datasets.

normalized and unnormalised datasets[7]. It can be seen that normalizing the values of the UPM feature has revealed a difference in the median values between the positive and negative examples of the Adapter role. The opposite has, however, happened in the case of the AIADU feature in which normalizing its values has reduced the difference between the distributions of the positive and negative instances. This explains why this feature, and some other intuitively relevant features, have relatively low ranks in normalized DP datasets, as mentioned in the previous paragraph. It also indicates that the relevance of some features can, in some cases, be diminished by normalization, and this (negative) consequence of normalization had not been anticipated.

### 6.2.2 Quality Impact

In this section, the impact of DPs on the quality of their participant classes are tested based on the quality metrics introduced in Chapter 4. The values of these quality metrics are calculated for the positive and negative examples of each role, and then the difference between these two groups of examples are tested (one

[7]The values in the unnormalised datasets have been min-max scaled to the range $[0,1]$ to better visualize the difference with the normalized values.

quality metric at a time) to determine if they are statistically different. In other words, for each role and each metric, the question of whether or not the positive and negative groups of examples come from the same distribution needs to be answered. The statistical test used to answer this question is the Mann-Whitney non-parametric test, which is the test used in the DP quality impact studies [23], [49] and [99], as discussed in Chapter 2. The confidence level used to determine statistical significance is 99% ($p - value < 0.01$).

Recalling that the neutral value 0.5 has been assigned in cases where the cohesion metrics are not calculable, as explained in Chapter 4, the tests are performed with and without these cases and the impact is reported as significant only if it is found to be so with and without them. This is to ensure that any detected statistical significance is not a result of the assignment of this neutral value. The impact on coupling, on the other hand, has been tested based on the normalized and unnormalized metrics in order to examine the effect of the threshold based normalization, which was proposed in Chapter 4, and it has been found to be the same in all but only four role-metric cases. This may be due to the fact that the size of the software systems, from which the datasets have been constructed, is widely diverse. The results reported are that of the normalized metrics.

The results of the statistical tests are shown in Table 6.3. While all of the significant impact identified for the Adapter, Composite and Command DPs are negative, the impact of the Decorator, Observer and Visitor DPs are mostly positive. This near uniformity of impact analysis results of individual DPs is a surprise as it is conventionally expected that different DPs improve and deteriorate different quality aspects [7]. Also, although the first consequence listed for the Command DP in [71] is that it decouples the class that requests the execution of an operation from the one that actually executes it, the class participants of its implemented instances are found to be coupled to significantly more other classes than non

Table 6.3 DP impact on software quality.

| DP | Roles | Complexity | | | | Cohesion | | | | Coupling | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | WMC | WMCmax | WMCavg | WMCstd | AMC | CMC | PMC | AAC | IAC | ICC | IIC | IDC | AEC |
| Adapter | Adaptee | R | R | G | R | R | R | R | G | G | R | R | R | R |
| | Adapter | R | R | R | R | G | G | R | G | R | R | R | R | G |
| Composite | Component | G | G | G | G | G | G | G | G | G | G | G | G | R |
| | Composite | R | G | G | G | G | G | R | G | R | G | G | R | R |
| Command | Receiver | R | G | G | G | G | G | R | G | R | G | G | R | R |
| | C. Command | R | R | G | R | R | G | R | G | R | R | R | R | R |
| Decorator | Component | G | Gr | Gr | Gr | G | G | G | G | G | G | Gr | G | R |
| | Decorator | G | Gr | Gr | Gr | Gr | Gr | G | Gr | R | Gr | Gr | R | Gr |
| Observer | Subject | R | G | Gr | Gr | G | G | R | G | R | G | G | R | R |
| | Observer | Gr | Gr | Gr | Gr | G | G | G | G | R | Gr | Gr | G | R |
| Visitor | C. Element | Gr | Gr | Gr | Gr | Gr | G | Gr | G | G | Gr | Gr | Gr | R |
| | Visitor | G | Gr | Gr | Gr | G | G | G | G | Gr | G | Gr | R | G |
| Proxy | RealSubject | G | G | G | G | G | R | G | G | G | G | G | G | R |
| | Proxy | G | G | G | G | G | G | R | G | G | G | G | R | G |

*__Green:__ positive impact. __Red:__ negative impact. __Grey:__ no (significant) difference.

participant classes. Moreover, import coupling at design level (as measured by the *IDC* metric) is significantly higher in classes playing all but one role while, on the other hand, their coupling at implementation level (as measured by the *IIC* metric) is significantly lower in more than half the significant cases. This is contrary to what would be expected given that DPs are defined at the design level and aim mainly to reduce coupling.

There are, however, results that match expectations, the most interesting of which is the implementation of the *program to an interface not an implementation* design principle by classes playing the Decorator and Observer roles, which is

demonstrated by having significantly more abstract couplings (as measured by the *IAC* metric) and significantly less concrete couplings (as measured by the *ICC* metric). Also, in Decorator DP instances, Decorator classes are supposed to be invisible to clients classes, and the results show that is actually the case (i.e. their export coupling, as measured by the *AEC* metric, is significantly low). Although there is no significant difference in the overall complexity (as measured by the *WMC* metric) of classes playing Decorator roles, the complexity is better distributed within these classes than within other classes, which is demonstrated by having significantly lower values in the *WMCavg* and *WMCstd* metrics. The same applies for the Observer and Visitor DPs although some of their roles also have significantly better or worse overall complexity.

The Adapter roles have been found to have significant impact on the distribution of more metrics than the roles of other DPs, which is probably due to the statistical power provided by their relatively large datasets. The roles of the Proxy DP, on the other hand, have been found to have significant impact on the least number of metrics (i.e. two metrics for each role). This may be a consequence of having lower minimum vote for positive examples during the construction of its datasets, as discussed in Chapter 5, which may have resulted in producing less accurate datasets in which many false positives have been added as positive examples.

The quality metrics that capture a statistically significant impact of a DP role will be added to the set of input features in the classification model trained to recognize the role, which will further enrich the feature subsets selected in the previous subsection. While the feature subsets in the previous subsection have been selected by using mutual information criterion of relevance, it can be said that Mann-Whitney test has been used in this subsection to asses feature relevance with a threshold set on the p-value, which is a perfectly acceptable feature selection approach [54].

In the following sections, the process of training the classification models (with and without adding the quality metrics) will be discussed and conclusions will be drawn with regard to the effect of including the quality metrics on the recognition accuracy.

## 6.3 Classification Model

### 6.3.1 Machine Learning and Classification

As discussed earlier, the problem of DP recognition is primarily a problem of inferring the design intent of potential DP instances, which is too complex to be solved by a set of few manually derived recognition rules, if any such rules can ever be known in the first place. However, even if the intent was not to be considered, the problem of drawing a line between instances of DP variants and other random sets of interconnected classes, based on how similar/different their interrelationships are to the DP, would still be difficult. Nevertheless, these challenging problems of DP recognition lend themselves naturally to solutions provided by machine learning for the reason explained in the next paragraph.

In order to classify a class or a set of interconnected classes into true and false categories denoting whether or not they are instances of a particular concept (i.e. a DP or a DP role), what is needed is a mapping function $g(.)$ that, given a potential instance represented as a feature vector $X$, it returns a category label $Y$. The fact that these complex mapping functions are not known, and that there exist a set of positive and negative examples in the training datasets prepared for each concept, makes it a typical situation in which machine learning can be used to approximate the functions based on their corresponding training datasets [4]. The approximation of these functions (or, in other words, the induction of the classification models)

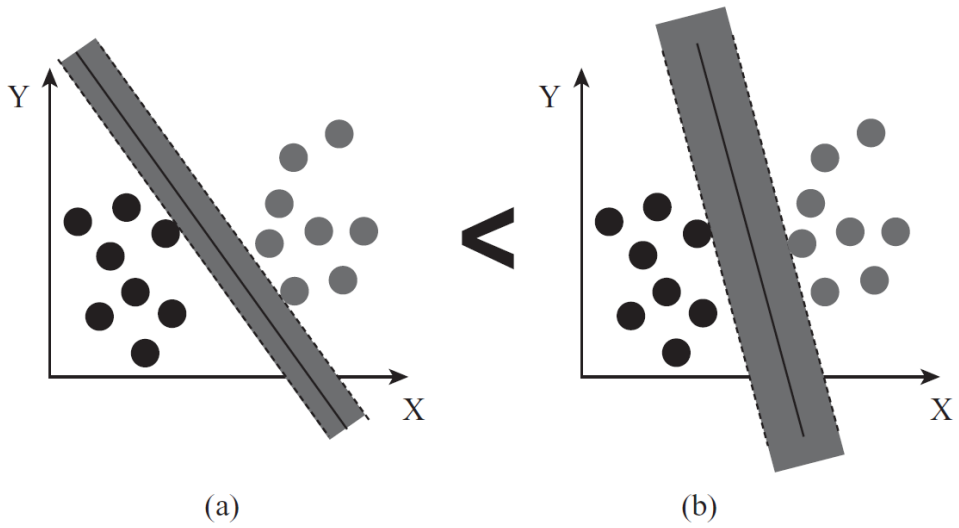



Fig. 6.4 Separating Hyperplanes [96]: (a) with small margin (b) with large margin.

involves positing the general form of a model and then estimating its unknown parameters based on the datasets [57].

The machine learning method used to induce the classification models is the Support Vector Machine (SVM) [40], and this selection is motivated by its wide popularity and practical success in many application areas [35] [166] [4]. There are also three other reasons for selecting SVM [96] [151]. First, it has only two parameters to optimise (discussed below), which makes finding the optimal combination of parameter values easy compared to ANN, for example. Second, SVM guarantees finding the unique optimal solution. The third reason is the fact that SVM is an implementation of the structural risk minimization (SRM) principle, which is known to lead to good generalization performance.

The main idea of SVM is to find the optimal decision hyperplane that separates the positive and negative examples with the maximum margin (Fig. 6.4). As suggested by the statistical learning theory, maximizing this margin improves the generalization performance of the induced models [96]. For a training dataset $D = \{(x_1, y_1), ..., (x_l, y_l)\}$ with $l$ training examples, where $x_i \in R^n$ is an $n$-dimensional

input feature vector and $y_i \in \{+1, -1\}$ is its target label, the linear SVM classifier searches for an optimal separating hyperplane:

$$\mathbf{w}^\intercal \mathbf{x} + \mathbf{b} = \mathbf{0} \tag{6.6}$$

where $\mathbf{w}$ is a weight vector and $\mathbf{b}$ is a scalar bias. The hyperplane defined by Eq. (6.6) lies halfway between two bounding hyperplanes given by:

$$\mathbf{w}^\intercal \mathbf{x} + \mathbf{b} = +\mathbf{1} \text{ and } \mathbf{w}^\intercal \mathbf{x} + \mathbf{b} = -\mathbf{1} \tag{6.7}$$

Therefore, the margin equals half the distance between these two bounding hyperplanes (i.e. $\frac{1}{\|\mathbf{w}\|_2}$), which is the parameter to be maximized [35]. However, in real world applications, including the one at hand, there may be no hyperplane that perfectly separates the two example classes (i.e. positive and negative), and so, the constraint that all data points have to be in the correct side of the hyperplane need to be *soften*. More complex nonlinear models may also be required in order to optimally separate the classes and, in such a case, the *input space* needs to be mapped into a higher dimensional *feature space* in which the separating hyperplane can be defined. This mapping can be performed by using a *kernel* function. Searching for the optimal hyperplane can now be formulated as the following optimization problem [90]:

$$\begin{aligned} \min_{\mathbf{w},\mathbf{b}} \quad & \frac{1}{2}\mathbf{w}^T\mathbf{w} + \mathbf{C}\sum_{i=1}^{l}\xi_i \\ \text{subject to} \quad & y_i(\,\mathbf{w}^T\phi(\mathbf{x_i}) + \mathbf{b}\,) \geq \mathbf{1} - \xi_i, \xi_i \geq 0 \end{aligned} \tag{6.8}$$

where $\phi()$ is the *kernel* function, which maps training vectors $x_i$ into a higher dimensional space, $\xi_i$ is the distance between the margin and a misclassified training data point and $C$ is a configurable parameter that represents the cost of this

training error. The parameter $C$ controls the trade-off between the minimization of training error and the maximization of the margin. Using too small $C$ values for nonlinear classification problems places insufficient stress to fit the training datasets which leads to under-fitted classification models. For classes that are linearly separable, on the other hand, using a large $C$ value may lead to over-fitting the training data.

The kernel function used is the nonlinear Radial Basis Function (RBF) which is, as suggested in [90] and [96], a reasonable first choice. The RBF is defined as follows:

$$K(x_i, x_j) = exp(-\gamma \|x_i - x_j\|^2), \gamma \geq 0 \tag{6.9}$$

This kernel function has one parameter (i.e. $\gamma$) which determines the non-linearity degree of kernel transformation. Similarly to the $C$ parameter, large $\gamma$ values leads to more complex non-linear models, and vice versa. The linear kernel can, in fact, be seen as a special case of the RBF kernel and, as noted in [98], conducting a complete model selection search with the RBF kernel (i.e. searching for a good pair of $C$ and $\gamma$ values) eliminates the need for considering the linear one. This wide range of complexity options may be particularly required for the classification problems at hand because individual DPs and DP roles are probably different in terms of their complexity. A two-stage (coarse and fine) grid search will be conducted to determine a good pair of parameter values for each one, the procedure of which is discussed in the following subsection.

## 6.3.2 Model Selection and Training

### 6.3.2.1 Training Dataset Preprocessing

An essential step, before starting the procedure of model selection and training, is to preprocess the training datasets based on which models are to be selected

and trained. This step can potentially have a significant positive impact on the generalization performance of the induced classification models [97]. The data preprocessing performed consists of the following steps:

- **Min-Max Scaling:** The first preprocessing performed is scaling feature values into the range $[0, 1]$ using min-max scaling, and this is performed only on the unnormalized versions of the datasets as the normalized ones are already in this range. Scaling is a very important step mainly because it avoids the problem of having features with large ranges of values dominating those with smaller ranges [90]. The scaling factors are saved alongside the trained models in order to be used later to scale the testing datasets.

- **Removing Redundant Instances:** Reducing the dimensionality of the training datasets to only the subsets of features selected in the previous section renders some of their instances to be identical ones. Since cross validation will be used for model selection (discussed below), it is necessary to remove any duplicated/redundant instances or, otherwise, the same instances may end up in the training and validation subsets, which leads to inaccurate estimations of the generalization performance[8]. Reducing the number of training examples by removing redundant ones also helps to reduce the training time which is highly dependent on the size of the training datasets [171].

- **Removing Contradictory Instances:** An interesting observation is that reducing the dimensionality of the datasets has revealed some contradictory training examples (i.e. having the same values in the input feature vector but with different target labels). Recalling the way in which the datasets have been constructed (in Chapter 5), it can be said that some of the DP instances

[8] In the case of the Adapter DP, only about 15% of the $305,811$ negative examples have non-zero feature vectors due the issue explained in Chapter 5 (Section 5.4.5). The number of *unique* negative instances for this DP after reducing the dimensionality is $12,347$.

that have been voted for by a single DP recognition tool, and hence used as negative examples, are probably positive ones but are somehow obscured and difficult to recognize. Representing these DP instances as training instances with only the relevant features has uncovered their true nature. It can also be said that the particular set of feature values in these contradictory training examples have actually received an additional vote, besides the ones received during dataset construction, for being positive examples. So, all of the contradictory examples have been resolved by only keeping the positive ones[9].

- **Oversampling:** The training datasets prepared for DPs and DP roles are highly imbalanced with about 50 and 30 times (on average, respectively) more negative instances than positive ones, and such imbalanced datasets can significantly affect the performance of most learning algorithms including SVM [86]. Several possible explanations have been proposed for the sensitivity of SVM to the imbalance in training datasets. One explanation is that the density of the majority (negative) instances around the borderline area where the ideal separating hyperplane should pass will be higher than the density of minority (positive) instances. The hyperplane is, consequently, shifted further away from the majority instances in order to reduce the misclassification cost. A number of methods have been developed to solve this problem for SVM, and they generally fall into two categories: data preprocessing methods (e.g. oversampling) and algorithmic modifications methods [19].

  The method used is Adaptive Synthetic (ADASYN) oversampling method [85]. The ADASYN method adaptively synthesizes different number of synthetic instances between each minority instance and its minority neighbours. The

[9]The contradictory negative examples have also been removed from the dataset used in Section 6.2 to evaluate the quality impact of DPs.

**Algorithm 1** Modified ADASYN

**procedure** ADASYN($D$)
  $d = \frac{|D_{min}|}{|D_{maj}|}$
  **if** $d < d_{th}$ **then** ▷ $d_{th}$ is a threshold for the tolerated degree of imbalance.
    $G = (|D_{maj}| - |D_{min}|) \times \beta$ ▷ $G$ is the total number of instances to be synthesized. ▷ $\beta$ specifies how much of the gap is to be reduced.
    **for** $s_i \in D_{min}$ **do**
      $kNN[i] = k$ nearest neighbours of $s_i$ in $D$ ▷ $k$ is set to 20.
      **if** $|kNN[i]_{maj}|$ equals to $|kNN[i]|$ **then**
        $r[i] = 0$
      **else** ▷ $kNN[i]_{maj}$ is the subset of majority neighbours.
        $r[i] = \frac{|kNN[i]_{maj}|}{|kNN[i]|}$
      **end if**
    **end for**

    **for** $j = 1, ..., |D_{min}|$ **do**
      $r^*[j] = \frac{r[j]}{\sum_{i=1}^{|D_{min}|} r[i]}$
      $g[j] = r^*[j] \times G$
    **end for**

    **for** $j = 1, ..., |D_{min}|$ **do**
      **for** $n = 1, ..., g[j]$ **do**
        $s_{jn} =$ randomly choose a minority nearest neighbour for $s_j$ from $kNN[j]$
        $new+ = s_j + (s_{jn} - s_j) \times \lambda$
      **end for** ▷ $\lambda$ is a random number in the range $(0, 1)$
    **end for**

  **end if**
**return** *new*
**end procedure**

more majority neighbouring instances a minority instance has (or, in other words, the closer a minority instance is to the *borderline*), the more instances are synthesized based on it. The fact that it focuses and synthesizes more instances on the borderline area, where the ideal SVM hyperplane should pass as explained above, is one of the reasons why ADASYN is selected. Another reason is that it can adaptively synthesize varying amounts of instances based on the degree of imbalance and the level of balance required.

There is, however, one problem with ADASYN method. While it creates the largest number of synthetic instances in cases where all neighbours are from

the majority, such cases are considered in Borderline-SMOTE[10] method [83], for example, to be noise for which no synthetic instances are created. So, ADASYN is slightly modified in this thesis to exclude such noisy instances from the oversampling process, which could otherwise degrade the quality of the produced training datasets. Algorithm 1 shows the modified ADASYN[11]. The input of the algorithm is an imbalanced dataset $D$ with $m$ training instances ($|D_{min}|$ minority instances and $|D_{maj}|$ majority instances). In the algorithm, the threshold $d_{th}$ is set to 0.90 although this has no effect as the imbalance in all datasets is much greater, and the parameter $\beta$ is set to 1 in order to produce fully balanced datasets.

#### 6.3.2.2 Model Selection

Since, for each DP and DP role, neither the best pair of parameter values (i.e. $C$ and $\gamma$) nor the exact range on which the search for the best pair needs to be focused, best pairs are exhaustively grid-searched in the parameter space. During the search, the generalization performance are estimated for each model (i.e. pair of parameter values) by using 5-times repeated 5-fold stratified cross-validation (with a split of 20% testing and 80% training), which gives more reliable estimates than if a single cross-validation run is used [167]. Before each cross-validation run, the prepared and preprocessed datasets are randomly divided into five subsets, which are then used to estimate the classification error rate following the cross validation procedure. The error rates of the five runs are averaged, and the final models are trained on the whole datasets using the pairs of parameter values that have led to the best average. It is these final models that are used to conduct the tests discussed in the next section.

[10]Synthetic Minority Oversampling Technique.
[11]The only modification done is the *if* statement in line 7.

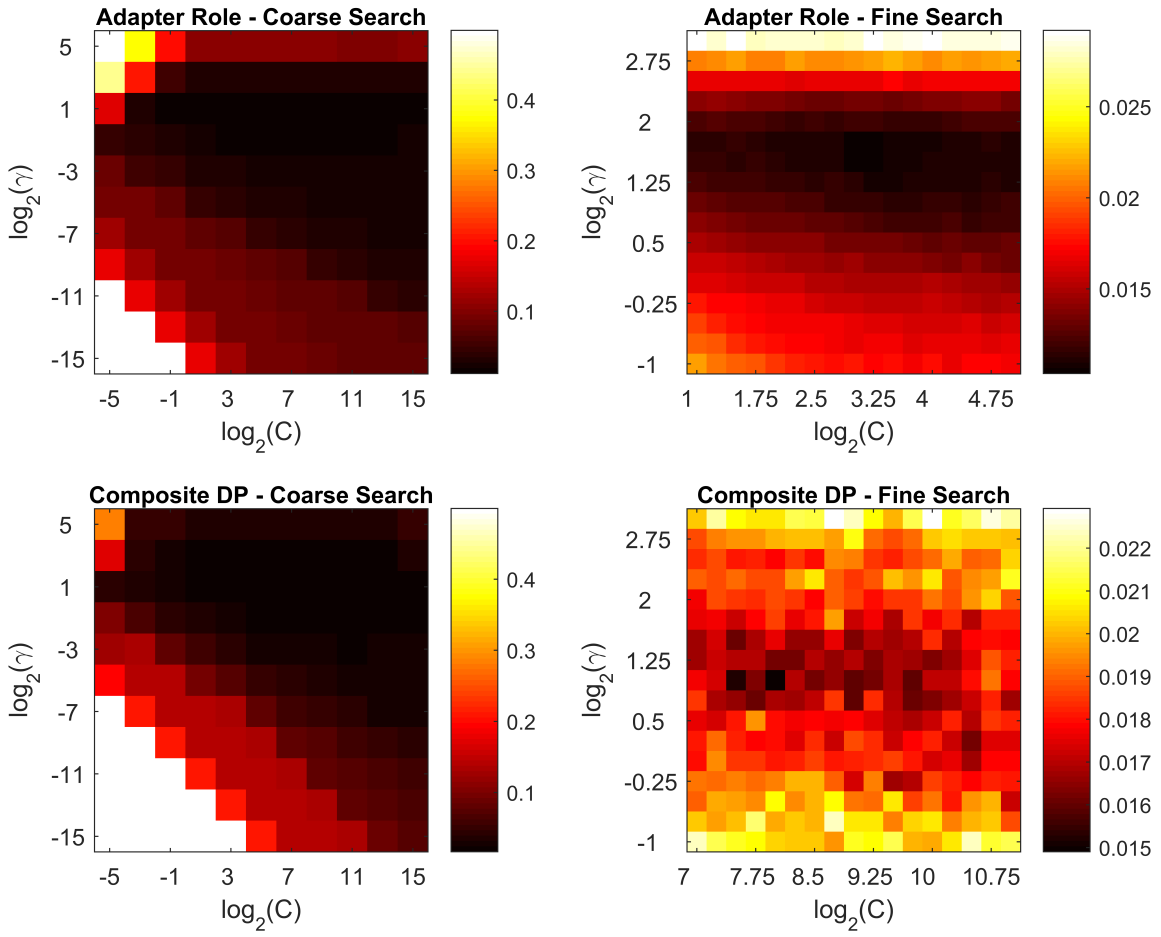



Fig. 6.5 Examples of error rates during the two (coarse and fine) stages of the grid-search.

Performing an exhaustive grid-search in one go is, however, time consuming given the large number of possible parameter value combinations. So, as recommended in [90], the grid-search is performed in two (coarse and fine) stages. In the first stage, the parameter values are coarsely spaced with $C = \{2^{-5}, 2^{-3}, \ldots, 2^{15}\}$ and $\gamma = \{2^{-15}, 2^{-13}, \ldots, 2^{5}\}$. In the second stage, a finer grid-search is conducted around the area of the best pair identified in the preceding stage (i.e. from $-2$ to $+2$ of the exponents with exponent increments of 0.25). Examples for the classification error rates estimated for different combinations of parameter values during the two stages of the grid-search are shown in Fig. 6.5.

## 6.4 Evaluation and Comparison

This section presents and discusses the experiments conducted to test the classifiers trained in the previous section. By testing these classifiers, conclusions with regard to the following issues can be drawn:

- Whether or not the design information captured by the feature set introduced in Chapter 3 is adequate for the purpose of DP recognition, and which of the feature calculation methods (i.e. absolute or context-based normalised values) leads to better recognition accuracy.

- The adequacy of the structures of DP and DP role training instances specified in Section 6.1.3 and whether or not the features added to their (global) input feature vectors in Section 6.1.4 provide the information required to perform the task of their respective phases.

- Whether or not including subsets of the quality features introduced in Chapter 4, which have been found in Section 6.2.2 to capture the quality impact of DP roles, improves their recognition accuracy.

- Whether or not the ACODD approach to DP dataset construction, which was introduced in Chapter 5, does produce training datasets with sufficient quality that can lead to better recognition accuracy than the tools used in their construction.

The training and testing of the SVM models have been implemented by using the Matlab interface of LIBSVM [36], which is one of the most widely used library for SVMs. The following subsection describes the experimental setup, and the results are then presented in Section 6.4.2 and discussed in Section 6.4.3.

### 6.4.1 Experimental Setup

#### 6.4.1.1 Test Subjects

The subjects to be tested are the classifiers trained to recognize DPs and DP roles, which are used to implement, respectively, phase two and phase one of the recognition process. Four different classifiers are trained for each DP role by using absolute and normalized versions of their training datasets, with and without including the quality features. Only two (i.e. absolute and normalised) versions of phase-two (i.e. DP) classifiers are trained, each of which is tested twice with and without including the quality features in the preceding phase. Besides the conclusions to be drawn based on comparing the accuracy of different classifier versions, these tests will be in a way like a search for the best configuration to implement the recognition system.

Phase-one classifiers are tested on all of the classes of the test software systems, and their accuracy will be evaluated based on how well they reduce the search space while maintaining a preferably perfect recall. Phase-two classifiers, on the other hand, are tested on all combinations of related roles' candidates that are passed by the preceding phase, and their accuracy will be evaluated based on how well they distill and recognize true DP instances and discard all other combinations of candidates. For more thorough and comprehensive testing of phase-two classifiers, the decision threshold of phase-one classifiers is moved towards the negative cases so that more classes will be falsely recognized as candidates and passed as test classes to phase-two classifiers. Also, classes that are mistakenly missed by phase-one classifiers, if any, will be added to the test set of the classifiers in the following phase for a more accurate assessment of their accuracy.

Since the costs of misclassifying positive and negative candidates in phase one are not equal (i.e. one would prefer a classifier that has a perfect recall even at the cost of misclassifying many negative candidates as positive ones), *threshold-moving*

is not, in fact, only beneficial for phase-two testing as explained above, but it is also useful for the task and purpose of phase-one itself. This is because, in such cases of unequal misclassification costs, techniques like *threshold-moving* as well as modifying the probability estimates of classifiers should be and are normally used [177] [105].

Although the default label output of SVMs is the sign of the distance from the separating hyperplane, the SVM library used provides an option to produce probability estimates, which are approximated by fitting a sigmoid function on the distance scores [110]. It is these probability estimates on which the decision threshold is set and it is chosen to be 0.01. The choice of this particular threshold is inspired by the significance level in hypothesis testing, one common value of which is the afore-chosen threshold. So, it can be analogously said that the null hypothesis in phase one is: *a class is playing a role in a DP*, which is rejected only when the estimated probability is less than the threshold. It is needless to say that no statistical tests are performed at this stage.

With such setting, the probability of having a perfect recall increases and the search space is reduced by filtering out classes that are highly unlikely to be role-playing classes (as deemed by the trained classifiers), which are exactly the objectives set for phase one as discussed in Section 6.1.1. The accuracy evaluation results of phase-one classifiers will, nevertheless, be presented for two cases: when the threshold is set on estimated probabilities and when the default label output is used. For phase-two classifiers, however, the results will be presented only when the default label output is used.

#### 6.4.1.2 Benchmark Systems

For an objective and independent assessment of recognition accuracy, the accuracy evaluation will be based on the peer-reviewed P-MARt repository. This repository

has been prepared by repetitive analysis of a few systems which has been performed by different teams [77], as mentioned earlier. Although it may not be a *perfect* gold standard benchmark, the fact that it is a peer-reviewed repository, which has been developed for the purpose of serving as a "baseline to assess the precision and recall of pattern identification tools" [73, page 1], makes it the best available benchmark.

Although the P-MARt repository contains DP instances from 9 open-source software systems, there are only 7 systems with instances for the DPs considered in this thesis. One of these systems is *Netbeans* $v1.0.x$ which is, however, excluded for two reasons. First, individual Adapter instances are composed of multiple role-playing classes for each of the Adapter and Adaptee roles, and it is difficult to objectively identify which Adapter class is adapting which Adaptee class. Second, a comment is placed in place of the Observer instances in the repository stating that many of the role-playing classes in the Adapter instances are also playing roles in Observer instances, without giving any details about the supposedly existing Observer instances. The remaining 6 systems are used as test systems[12], and table 6.4 shows some information about them and the total number of DP instances they include.

It should be noted that the benchmark instances that include role-playing classes from external software libraries have been removed, and all the remaining instances are used as a reference list for the accuracy evaluation. Any recognized instance that is or is not in this reference list will be counted, respectively, as a true and false positive instance, and any missed reference instance will be counted as a false negative instance.

[12]The list of the 539 software systems from which the training datasets have been constructed does not include any of the test systems, and not even any of their other versions.

Table 6.4 Test software systems.

| Test System | # of Classes | # of Role-Playing Classes* | # of DP Instances |
|---|---|---|---|
| **JHotDraw v5.1** | 155 | 54 | 56 |
| **JRefactory v2.6.24** | 569 | 128 | 115 |
| **JUnit v3.7** | 78 | 8 | 6 |
| **MapperXML v1.9.7** | 175 | 6 | 5 |
| **PMD v1.8** | 282 | 100 | 94 |
| **QuickUML v2001** | 155 | 4 | 2 |

* Some classes are playing different/same roles in multiple DP instances.

#### 6.4.1.3 Evaluation Measures

The evaluation measures which will be used to evaluate the accuracy of phase-one and phase-two classifiers differ according to the objectives of each phase. Since the task of phase one is (1) to filter out as many non role-playing classes as possible and (2) to recognize and pass all role-playing classes, the two measures used in this phase are the *Specificity* and *Recall* as they respectively capture the performance with regards to the aforementioned objectives. These two measures can be calculated based on the number of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) as follows:

$$Specificity = \frac{TN}{TN + FP} \tag{6.10}$$

$$Recall = \frac{TP}{TP + FN} \tag{6.11}$$

The G-mean measure will also be used as it combines the specificity and recall values into one value as follows:

$$G\text{-}mean = \sqrt{Specificity \times Recall} \tag{6.12}$$

As for the second phase, since it is the phase that produces the final output results of the recognition system, the objective is to have results that are complete (i.e. include all existing true DP instances) and precise (i.e. do not include any or, at least, not many false positives). While the completeness is measured by the *Recall*, the preciseness is measured by the *Precision* measure which can be calculated as follows:

$$Precision = \frac{TP}{TP + FP} \tag{6.13}$$

The recall and precision values can also be combined into a single value by using the F-Measure, which can be calculated as follows:

$$F\text{-}Measure = \frac{2 \times Recall \times Precision}{Recall + Precision} \tag{6.14}$$

Although the accuracy measures above do give an insight into how well the trained classifiers have performed, the values of these measures need to be put into perspective by comparing them with the values calculated for other recognition tools. However, the evaluation results reported for different tools in the literature are based on different set of test systems and different (mostly subjective) reference set of instances. Also, different *occurrence types* are used in different evaluations, which can have a large influence on the calculated precision and recall values and, in many cases, such measures are not even reported [132]. There are, moreover, differences in the way instances are aggregated and counted [169].

In order to avoid the problems described above and have an accurate and objective comparison with other tools, the tools need to be used to analyse the same set of test systems and their output results need to be transformed into instances with the same *occurrence type*, which are then evaluated based on the same reference set as the one used to evaluate the classifiers trained in this thesis. This can, however, be done with only the six tools used for the dataset construction

in Chapter 5 for the same reasons as explained in Section 5.4. The comparison with these tools is, nevertheless, of particular interest as it shows whether or not the training datasets produced by the ACODD approach can be of better accuracy than the tools used in their construction. In other words, it shows if the performance of the trainee can exceed that of the trainer.

Since phase one is basically a search space reduction phase, comparisons will be made based only on the output of phase-two classifiers. The six tools (henceforth, other classifiers) as well as the trained classifiers are first ranked based on the accuracy at which they recognize DPs, one DP at a time. The statistical significance of rank differences between the classifiers are then tested. Although the statistical test recommended in [48] for the task of comparing multiple classifiers over multiple (DP) datasets is the Friedman non-parametric test, the test used is Skillings–Mack (SM) test [37] which is equivalent to the former test but used for cases in which there are missing data[13]. The SM test evaluates whether or not there is a statistically significant difference in the observed rankings. The null-hypothesis to be tested is that all the classifiers are equivalent and any observed rankings differences are merely random. If this null hypothesis is rejected at $\alpha = 0.05$ level of significance, the Bonferroni-Dunn post-hoc test is then used to check if the classifiers trained in this thesis are better than the other classifiers. The post-hoc test used is also the one recommended in [48] for when a new classification method is compared to existing ones.

The comparison procedure described in the previous paragraph is repeated three times, one for each of the three measures used to evaluate the accuracy of phase-two classifiers.

[13]The missing data is due to the fact that two of the tools do not recognize some of the DPs.

### 6.4.2 Results

The following two subsections present the results of testing the accuracy of phase-one and phase-two classifiers, respectively. The accuracy differences between the trained classifiers and the other tools/classifiers are analysed and presented in Section 6.4.2.3.

#### 6.4.2.1 Phase One

Table 6.5 shows the results of evaluating the accuracy of phase-one classifiers as measured when the threshold of 0.01 is set on estimated probabilities and also when the default SVM label output is used. It can be seen that the specificity achieved for all DP roles, with all of their classifier versions, is over 90% when the default label output is used. This is still the case for the vast majority of classifier versions even when the threshold is used, and for most of the other versions, a minimum specificity of 77.3% (i.e. the specificity of the absolute-without quality version of the Adapter role classifiers) is achieved. Even at this minimum, 1064 classes out of the 1377 non role-playing classes have been correctly filtered out. Although a very low specificity is achieved with some classifier versions of the component roles (i.e. the Component roles of the Composite and Decorator DPs) as well as all of the classifier versions of the Proxy DP roles, using the default label output increases the specificity in these cases to over 97%.

With regards to their recall accuracy, all DP roles have a recall of 50% or greater ($\geq$ 75% in six roles) for at least one of their classifier versions, with the exceptions of three roles (i.e. the Adapter, Concrete-Command and Composite roles). Although the Concrete-Command role, for example, has a recall of only 23.1%, it has an almost perfect specificity of 98.9%. In plain numbers, out of the 1414 total number of test classes, only 19 classes were reported as role-playing candidates, 3 of which are indeed role-playing classes. These are the results when the threshold is used.

Table 6.5 The test accuracy performance of phase-one SVMs at two different decision thresholds.

| DP: Role | Feature Calculation Method | Specificity | | | | Recall | | | | G-mean | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | w/o Q* | | w/ Q* | | w/o Q | | w/ Q | | w/o Q | | w/ Q | |
| | | 0.01• | Def.⊙ | 0.01 | Def. | 0.01 | Def. | 0.01 | Def. | 0.01 | Def. | 0.01 | Def. |
| Adapter: **Adaptee** | **Absolute** | 0.844 | 0.912 | 0.868 | 0.955 | 0.391 | 0.304 | 0.565 | 0.130 | 0.575 | 0.527 | 0.701 | 0.353 |
| | **Normalized** | 0.914 | 0.958 | 0.942 | 0.975 | 0.304 | 0.174 | 0.304 | 0.087 | 0.527 | 0.408 | 0.535 | 0.291 |
| Adapter: **Adapter** | **Absolute** | 0.773 | 0.942 | 0.825 | 0.966 | 0.081 | 0.000 | 0.081 | 0.000 | 0.250 | 0.000 | 0.259 | 0.000 |
| | **Normalized** | 0.934 | 0.973 | 0.935 | 0.977 | 0.027 | 0.000 | 0.027 | 0.000 | 0.159 | 0.000 | 0.159 | 0.000 |
| Command: **Receiver** | **Absolute** | 0.966 | 0.986 | 0.987 | 0.992 | 0.750 | 0.750 | 0.500 | 0.250 | 0.851 | 0.860 | 0.703 | 0.498 |
| | **Normalized** | 0.994 | 0.995 | 0.999 | 1.000 | 0.750 | 0.750 | 0.500 | 0.250 | 0.863 | 0.864 | 0.707 | 0.500 |
| Command: **C.Command** | **Absolute** | 0.989 | 0.995 | 0.996 | 0.999 | 0.231 | 0.231 | 0.000 | 0.000 | 0.478 | 0.479 | 0.000 | 0.000 |
| | **Normalized** | 0.999 | 0.999 | 0.999 | 0.999 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| Composite: **Component** | **Absolute** | 0.965 | 0.994 | 0.972 | 0.994 | 0.667 | 0.500 | 0.333 | 0.333 | 0.802 | 0.705 | 0.569 | 0.576 |
| | **Normalized** | 0.053 | 0.996 | 0.000 | 0.996 | 0.833 | 0.500 | 0.833 | 0.333 | 0.211 | 0.706 | 0.000 | 0.576 |
| Composite: **Composite** | **Absolute** | 0.960 | 0.974 | 0.954 | 0.987 | 0.364 | 0.364 | 0.364 | 0.091 | 0.591 | 0.595 | 0.589 | 0.300 |
| | **Normalized** | 0.996 | 0.999 | 0.996 | 1.000 | 0.273 | 0.182 | 0.273 | 0.182 | 0.521 | 0.426 | 0.521 | 0.426 |
| Decorator: **Component** | **Absolute** | 0.963 | 0.990 | 0.986 | 0.992 | 1.000 | 1.000 | 1.000 | 1.000 | 0.981 | 0.995 | 0.993 | 0.996 |
| | **Normalized** | 0.979 | 0.989 | 0.001 | 0.994 | 1.000 | 1.000 | 1.000 | 1.000 | 0.989 | 0.995 | 0.038 | 0.997 |
| Decorator: **Decorator** | **Absolute** | 0.869 | 0.997 | 0.992 | 0.994 | 0.500 | 0.250 | 0.500 | 0.250 | 0.659 | 0.499 | 0.704 | 0.499 |
| | **Normalized** | 0.997 | 0.999 | 0.997 | 1.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| Observer: **Subject** | **Absolute** | 0.799 | 0.978 | 0.887 | 0.995 | 0.625 | 0.250 | 0.500 | 0.250 | 0.707 | 0.494 | 0.666 | 0.499 |
| | **Normalized** | 0.972 | 0.980 | 0.990 | 0.996 | 0.125 | 0.000 | 0.125 | 0.125 | 0.349 | 0.000 | 0.352 | 0.353 |
| Observer: **Observer** | **Absolute** | 0.969 | 0.971 | 0.968 | 0.977 | 0.545 | 0.545 | 0.545 | 0.545 | 0.727 | 0.728 | 0.727 | 0.730 |
| | **Normalized** | 0.965 | 0.967 | 0.974 | 0.984 | 0.545 | 0.545 | 0.545 | 0.455 | 0.726 | 0.726 | 0.729 | 0.669 |
| Visitor: **C.Element** | **Absolute** | 0.978 | 0.995 | 0.981 | 0.996 | 0.968 | 0.790 | 0.984 | 0.973 | 0.973 | 0.887 | 0.983 | 0.984 |
| | **Normalized** | 0.968 | 0.998 | 0.967 | 0.998 | 0.968 | 0.919 | 0.984 | 0.925 | 0.968 | 0.958 | 0.976 | 0.961 |
| Visitor: **Visitor** | **Absolute** | 0.990 | 0.991 | 0.989 | 0.991 | 1.000 | 1.000 | 1.000 | 1.000 | 0.995 | 0.995 | 0.995 | 0.995 |
| | **Normalized** | 0.865 | 0.962 | 0.836 | 0.967 | 1.000 | 1.000 | 1.000 | 1.000 | 0.930 | 0.981 | 0.914 | 0.984 |
| Proxy: **RealSubject** | **Absolute** | 0.000 | 0.970 | 0.001 | 0.974 | 1.000 | 0.000 | 1.000 | 1.000 | 0.000 | 0.000 | 0.027 | 0.987 |
| | **Normalized** | 0.000 | 0.996 | 0.001 | 0.998 | 1.000 | 0.000 | 1.000 | 0.000 | 0.000 | 0.000 | 0.027 | 0.000 |
| Proxy: **Proxy** | **Absolute** | 0.009 | 0.972 | 0.035 | 0.981 | 1.000 | 0.000 | 1.000 | 0.000 | 0.096 | 0.000 | 0.186 | 0.000 |
| | **Normalized** | 0.008 | 0.999 | 0.008 | 0.999 | 1.000 | 0.000 | 1.000 | 0.000 | 0.092 | 0.000 | 0.088 | 0.000 |

∗SVMs that are trained by datasets *without* and *with* the inclusion of quality features.
•The accuracy measures calculated with 0.01 threshold set on the estimated probability of being a role-playing class.
⊙ The accuracy measures with the default label output of SVMs.

While the best recall of 7 roles is maintained even when the default label output is used, it decreases in other roles with no significant specificity gains (i.e. $< 18\%$ in all but the case of the Component role of the Composite DP). The Proxy role is the only DP role that has role-playing classes recognized as candidates only when almost all of the classes are recognized as such, and hence it can be concluded that its trained classifiers have failed to learn the concept it represents[14].

Out of the 14 DP roles, 9 roles have a G-mean value of 70% or higher for at least one of their classifiers when the threshold is used. The normalized-with quality classifier version of the Observer role, for example, has a G-mean value of 72.9%, which is achieved as a result of correctly recognizing 6 out of the 11 role-playing classes while correctly discarding 1367 out of 1403 non role-playing classes. When the default label output is used instead, the G-mean of all cases in which the best recall rate is maintained (i.e. the 7 roles mentioned above) is increased driven by the increase in their specificity. For most of the other cases, the G-mean values are decreased as a result of having lower recall rates.

The bar charts in Fig. 6.6 visually depict the performance (when the threshold is used) of all the four versions of phase-one classifiers. The Proxy role is, however, excluded since all of its classifiers have failed to adequately recognize its role-playing classes, potentially for reasons that will be discussed later. It can be seen that the specificity is above 77% in all cases with the exceptions of some classifier versions of the Component roles as well as all of RealSubject classifiers, as noted earlier. However, unlike the RealSubject classifiers, the specificity of the low-performing Components' classifiers dramatically increases to about 99% by increasing the threshold to only 0.05, though at the cost of having lower recalls, as shown by the hollow triangles and circles in Fig. 6.6. The fact that most non role-playing classes have such low estimated probabilities, and hence the very high

[14]Because of this, the recall results of the Proxy role have not been considered or counted in the preceding sentences of this paragraph.

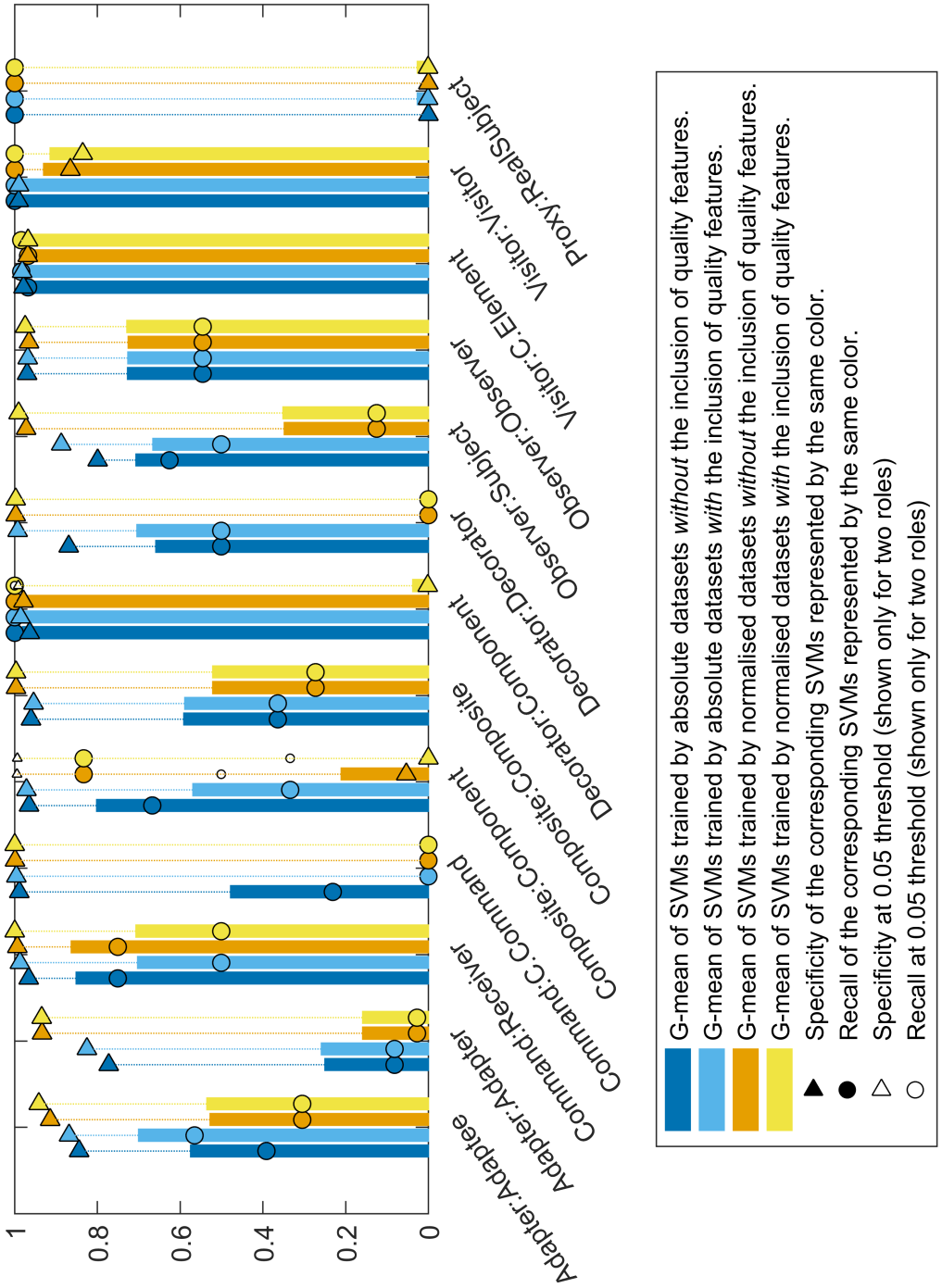



Fig. 6.6 The test accuracy performance of the SVMs trained for DP roles (phase-one SVMs).

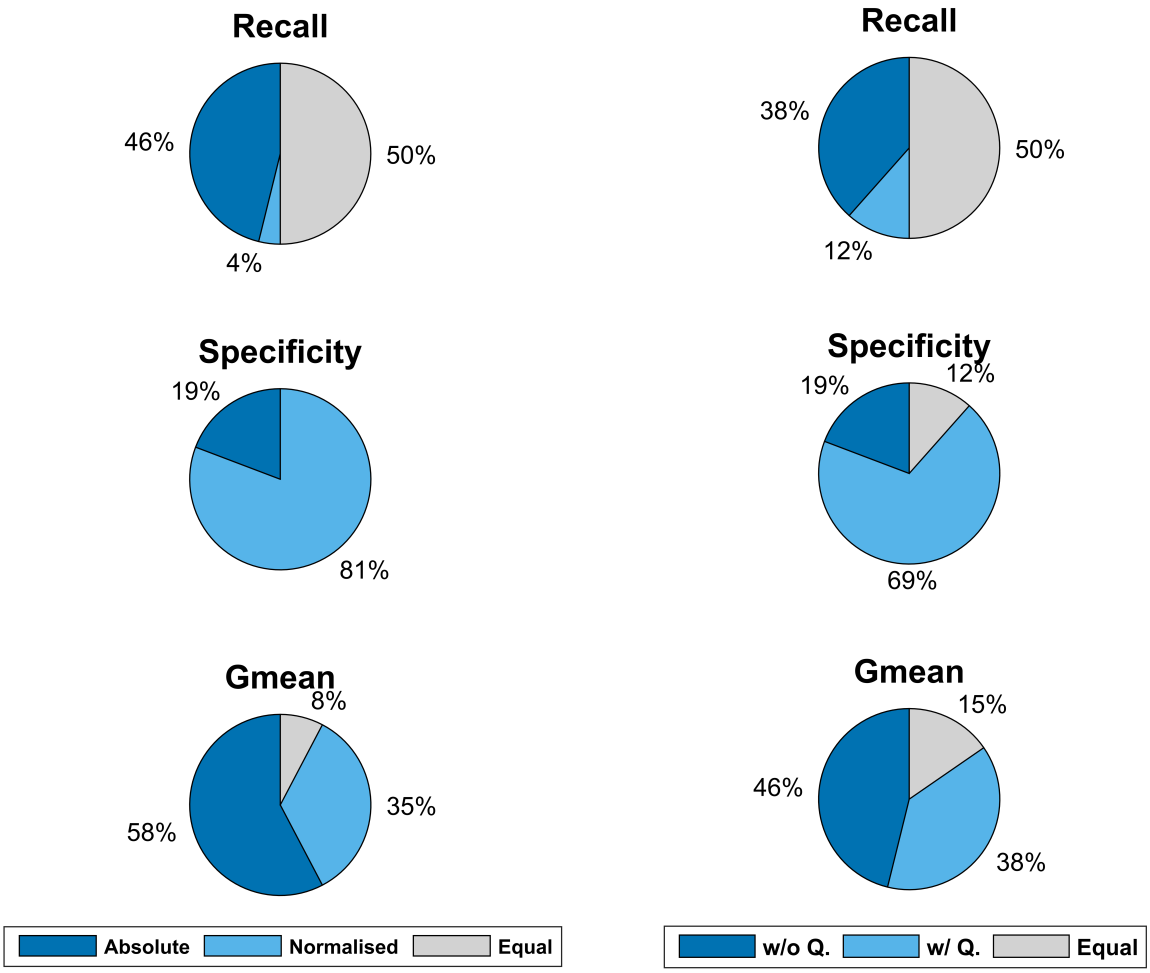



Fig. 6.7 Percentages of times a dataset type has led to better performance than the other.

specificity, as well as the fact that all but three roles have a best recall of at least 50%, shows how well the classifiers have learnt to discriminate between role-playing and non role-playing classes.

Having a good performance in at least one classifier version is enough to draw a conclusion that the corresponding role dataset, as constructed in Chapter 5, is of at least sufficient accuracy. The best performing classifier version of each role, according to the accuracy measure of interest, answers the questions of whether or not feature values should be contextually normalized and whether or not the quality features should be included. It is, as stated before, like a search for the best

configuration to be used for each role in the recognition system implementation. There are, however, certain types of datasets that have performed better than others more frequently, based on a given accuracy measure, across different DP roles. As can be seen in Fig. 6.7, while context-based normalization of feature values have led to better specificity performance in 81% of the times, using the absolute feature values is more likely to give better performance in terms of the recall and G-mean measures. The effect of including the quality features is similar to that of the context-based normalization.

#### 6.4.2.2 Phase Two

As explained in the experimental setup, the test sets for DP classifiers are formed by all the combinations of role-playing candidates that are passed by the classifiers of the two corresponding roles. Although, as specified in the setup, the threshold of 0.01 is set on the probabilities estimated by role classifiers for more thorough and comprehensive testing of DP classifiers, the default label output will be used in the case of the Proxy DP roles. This is because of the failure of the Proxy role classifiers and the likely failure of the Proxy DP classifiers as well for the same reason as going to be discussed later. Despite the fact that this gives an advantage to the classifiers of this DP as they would be subjected to far less negative DP instances, they have failed as expected (i.e. they have zero precision and recall), and hence this DP will not be considered further in this subsection.

The results of testing the accuracy of DP classifiers are shown in Fig. 6.8. With the only exception of the Adapter DP, every DP has been recognized with a precision of 70% or greater by at least one of its two classifiers. As can be seen in the figure, precision is usually better when the DP classifiers are preceded by with-quality version of phase-one classifiers, thanks to their better performance in specificity. Differences in terms of the raw number of FPs are, however, often

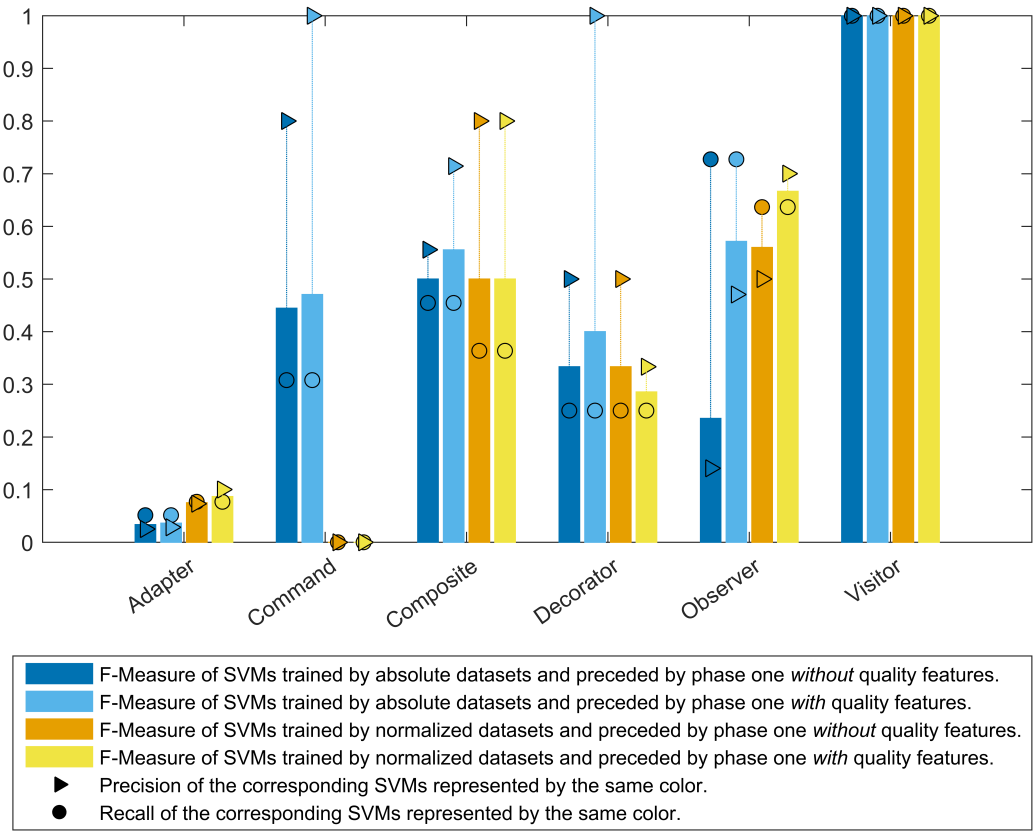



Fig. 6.8 The test accuracy performance of the SVMs trained for DPs (phase-two SVMs).

small. For example, while the precision of the absolute version of the Decorator DP classifiers is 100% when preceded by with-quality phase-one classifiers, the precision drops to 50% when preceded by without-quality classifiers. However, only one FP instance is recognized in the latter case besides a TP instance, and hence the 50% precision. This particular FP instance is recognized as a result of having nearly 2000 more potential DP instances to check, which shows the impressive precision of this DP classifier regardless of the classifiers used in the preceding phase. Nevertheless, the number of FP instances increases by 9, on average, for every 1000 additional potential instances passed by phase one. This shows that, as discussed in Section 6.1.1, breaking down the recognition process into two phases can improve the recognition accuracy.

The Adapter DP classifiers, as it is the case in precision, are the ones with the lowest recall rates. All of the other 5 DPs have been recognized with a recall of at least 25% (and, in two of which, the recall is $> 72\%$). The 25% recall minimum is achieved for the Decorator DP which have a total of only 4 positive instances in the test reference set. In fact, four of the seven DPs included in this experiment have 11 positive instances or less. Evaluating the recall ability of the trained classifiers with such a few positive instances may not provide an accurate recall assessment. However, the high precision at which the relatively low recall is achieved is an evidence that the trained classifiers have captured the concept. When the absolute version of the Composite DP classifiers, for example, is tested on 3668 potential instances, which include only 11 positive instances, the classifier has reported only five instances as positives, four of which are indeed true positives. This is tantamount to the task of searching for a needle in a haystack which have been, however, performed fairly successfully.

If the F-Measure values are rounded to one decimal point, four DPs will have an F-Measure value of 50% or greater. The ones that will not are the Adapter DP, given its low precision as well as recall, and the Decorator DP which has a high recognition precision but low recall rates. The DP that is recognized with surprisingly perfect precision and recall by both of its two classifier versions, is the Visitor. However, this perfect performance and the relatively poor performance of the Adapter DP as well as the failure of the Proxy DP can all be explained by looking at the performance of the other tools, which will be discussed in the following subsection. The table that shows the accuracy performance results of phase-two classifiers, which are used to produce Fig. 6.8, is provided in Appendix C.

#### 6.4.2.3 Comparison with Other Tools

Table 6.6 shows the results of evaluating the accuracy of the other DP recognition tools. It can be seen that all but the PINOT tool have failed when it comes to the recognition of the Proxy DP. While the DeMIMA, MARPLE and FINDER have not reported any Proxy instances at all, the SSA and WOP have reported only FP instances. Even in the only tool (i.e. PINOT) in which the Proxy DP is successfully recognized, it is recognized with a rather low precision of 1.6%. Also, since all of the Proxy instances reported by this tool are missing the names of the classes playing the RealSubject role, the benefit of the doubt is given to this tool, and so the reported instances are assumed to be correct as long as they got the name of the class playing the Proxy role right. Since all but one tool, as well as the classifiers trained in this thesis, have failed to recognize the Proxy DP, it will no longer be considered in the comparison.

The accuracy at which the Adapter DP is recognized, as shown in Table 6.6, is low for all of the tools with an average precision of 1.3%. The Visitor DP, on the other hand, have been accurately recognized by four of the six tools. Since the training datasets are constructed by using this same set of tools, if all of them have a poor recognition accuracy, as it is the case with the Adapter DP, the quality of the constructed dataset will consequently be negatively affected. The opposite is true if at least the majority of them are performing well, as it is the case with the Visitor DP, because this will prevent poorly performing tools from adding their FP instances to the constructed datasets as positive examples. This explains the performance of the classifiers trained for the two DPs as well as the failure of the Proxy classifiers. However, when put into the perspective of these other tools, the performance of the classifiers trained for the Adapter is not bad but, in fact, relatively better. Also, the perfect recognition accuracy of the classifiers trained

Table 6.6 The accuracy performance of the other DP recognition tools.

| DP | Measure | SSA | FINDER | MARPLE | PINOT | DeMIMA | WOP |
|---|---|---|---|---|---|---|---|
| **Adapter** | **Precision** | 0.025 | 0.000 | 0.021 | 0.000 | 0.009 | 0.023 |
| | **Recall** | 0.051 | 0.000 | 0.436 | 0.000 | 0.231 | 0.091 |
| | **F-Measure** | 0.034 | 0.000 | 0.040 | 0.000 | 0.017 | 0.037 |
| **Command** | **Precision** | 0.100 | 0.625 | 0.010 | X | 0.068 | X |
| | **Recall** | 0.308 | 0.192 | 0.885 | X | 0.654 | X |
| | **F-Measure** | 0.151 | 0.294 | 0.019 | X | 0.123 | X |
| **Composite** | **Precision** | 1.000 | 0.159 | 0.357 | 0.133 | 0.063 | 0.333 |
| | **Recall** | 0.273 | 0.636 | 0.455 | 0.200 | 0.273 | 0.182 |
| | **F-Measure** | 0.429 | 0.255 | 0.400 | 0.160 | 0.102 | 0.235 |
| **Decorator** | **Precision** | 0.400 | 0.750 | 0.033 | 0.500 | 0.025 | X |
| | **Recall** | 0.500 | 0.750 | 0.500 | 0.250 | 0.250 | X |
| | **F-Measure** | 0.444 | 0.750 | 0.063 | 0.333 | 0.045 | X |
| **Observer** | **Precision** | 0.353 | 0.000 | 0.020 | 0.014 | 0.067 | X |
| | **Recall** | 0.545 | 0.000 | 0.455 | 0.143 | 0.364 | X |
| | **F-Measure** | 0.429 | 0.000 | 0.038 | 0.026 | 0.113 | X |
| **Visitor** | **Precision** | 0.995 | 1.000 | 0.756 | 0.000 | 0.983 | 0.000 |
| | **Recall** | 1.000 | 1.000 | 1.000 | 0.000 | 0.935 | 0.000 |
| | **F-Measure** | 0.997 | 1.000 | 0.861 | 0.000 | 0.959 | 0.000 |
| **Proxy** | **Precision** | 0.000 | 0.000 | 0.000 | 0.016 | 0.000 | 0.000 |
| | **Recall** | 0.000 | 0.000 | 0.000 | 1.000 | 0.000 | 0.000 |
| | **F-Measure** | 0.000 | 0.000 | 0.000 | 0.032 | 0.000 | 0.000 |

**X** denotes that the tool does not attempt to recognize the DP.

for the Visitor DP is not actually surprising but a logical consequence of the high quality of its dataset.

While the other tools have performed generally better in terms of recall than the trained classifiers, they have worse performance in terms of precision as well as the F-Measure. The best configuration for the Observer DP, for example, leads to a classifier that performs better in terms of the two latter measures than the second best tool by about 35 and 24 percent points, respectively. Also, the classifier trained for the Command DP has precision and F-Measure rates that are higher by about

37 and 18 percent points, respectively, than its closest rival. This is particularly interesting given that one of the tools used to construct its training dataset (namely the SSA tool [160]) does not distinguish its instances from the instances of the Adapter DP, the classifier of which has also performed relatively better as noted in the previous paragraph.

On the other hand, although the MARPLE tool, for example, has better recall in the Adapter and the Command DPs, this is at the cost of recognizing 786 and 2398 FP instances, respectively. The advantage of the other tools in recall is generally smaller than the advantage of the trained classifiers in precision, and hence the higher F-Measure rates gained by the trained classifiers. The Decorator DP is, however, an exception in which the advantage of the FINDER and SSA tools in recall is greater than the advantage of the trained classifier in precision. It is worth noting that, although all of the DPs included in this experiment are claimed to be recognized by the SSA tool with a perfect precision and recall [160], this claim clearly does not hold true and this shows the extend to which subjective evaluation can lead to accuracy overestimation, as discussed in Section 2.2.10.8.

Fig.6.9 shows where the phase-two classifiers stands compared to the other classifiers/recognition tools based on precision, recall as well as the F-Measure. The results of the SM statistical test are also shown at the top of each sub-figure. The classifiers that are produced by the configurations that lead to the best accuracy performance, as measured by the F-Measure, are the ones selected for the comparison. Although, in some cases, better precision or recall performance have been achieved by other trained classifiers, the results of only those selected based on their F-Measure performance are used for consistency. As shown in the figure, the selected phase-two classifiers are ranked first based on the precision and the F-Measure in all but one DP. The differences between the classifiers' ranks in these two measures are statistically significant as demonstrated by the p-values. Although

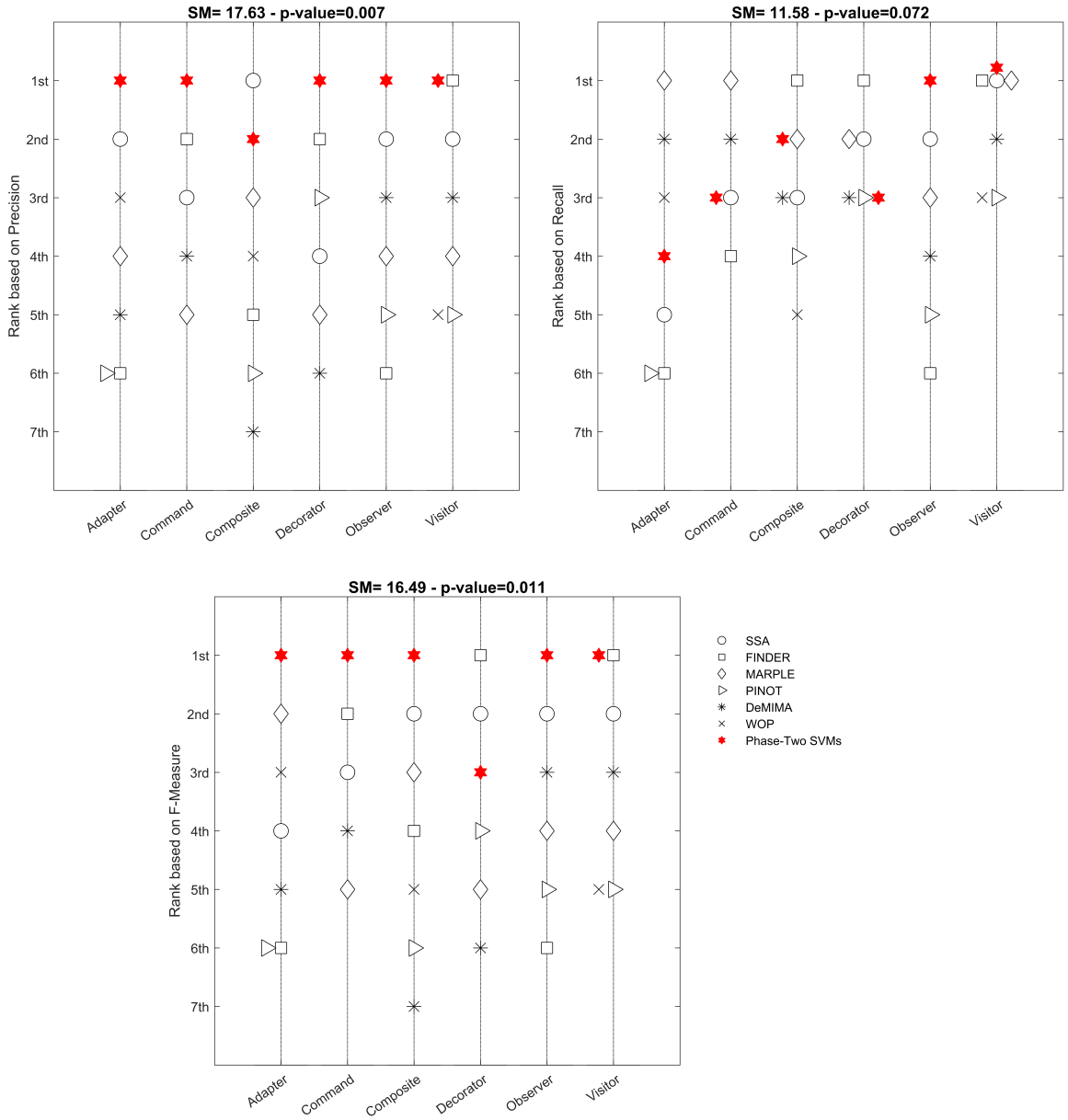



Fig. 6.9 Ranks of phase-two SVMs compared to the other recognition tools based on precision, recall and F-measure.

the phase-two classifiers have received relatively low ranks in recall, differences between the classifiers are not statistically significant.

In order to see how well the two-phase system performs as a whole (i.e. not only the phase-two classifiers), the test is repeated without adding role-playing classes that are mistakenly filtered out by phase-one classifiers. The ranks achieved by the system as a whole is shown in Fig. 6.10. The system generally maintains high precision ranks with a slight decline to the second rank in two DPs and a quite noticeable decline to the fourth rank in one DP. This decline in precision is not the result of an increased number of FP instances, but rather the result of a decrease in the number of TP ones as a result of filtering some role-playing classes out by phase-one classifiers. The statistical significance of the differences in precision ranks is, however, maintained. The same does not apply to the F-Measure in which the system has declined by about 2 rank steps on average and the differences between classifiers' ranks are no longer significant. The effect of the decrease in the number of TP instances can be seen in the the dramatic drop of the recall ranks, the differences in which has become significant, though not in the desirable direction given the system's low ranks.

The next step is use the Bonferroni-Dunn post-hoc test to assess the statistical significance of the differences between the classifiers trained in this thesis and the other tools. Since the post-hoc test should be applied only if the SM test indicates that a statistically significant difference exists, it is not going to be applied to test the differences between the phase-two classifiers and other tools in recall ranks nor the differences between the recognition system and the other tools in F-Measure ranks. The difference between the performance of a pair of classifiers is considered to be statistically significant if the averages of their ranks differ by at least the Critical Difference (CD):

$$CD = q_\alpha \sqrt{\frac{k(k+1)}{6N}} \tag{6.15}$$

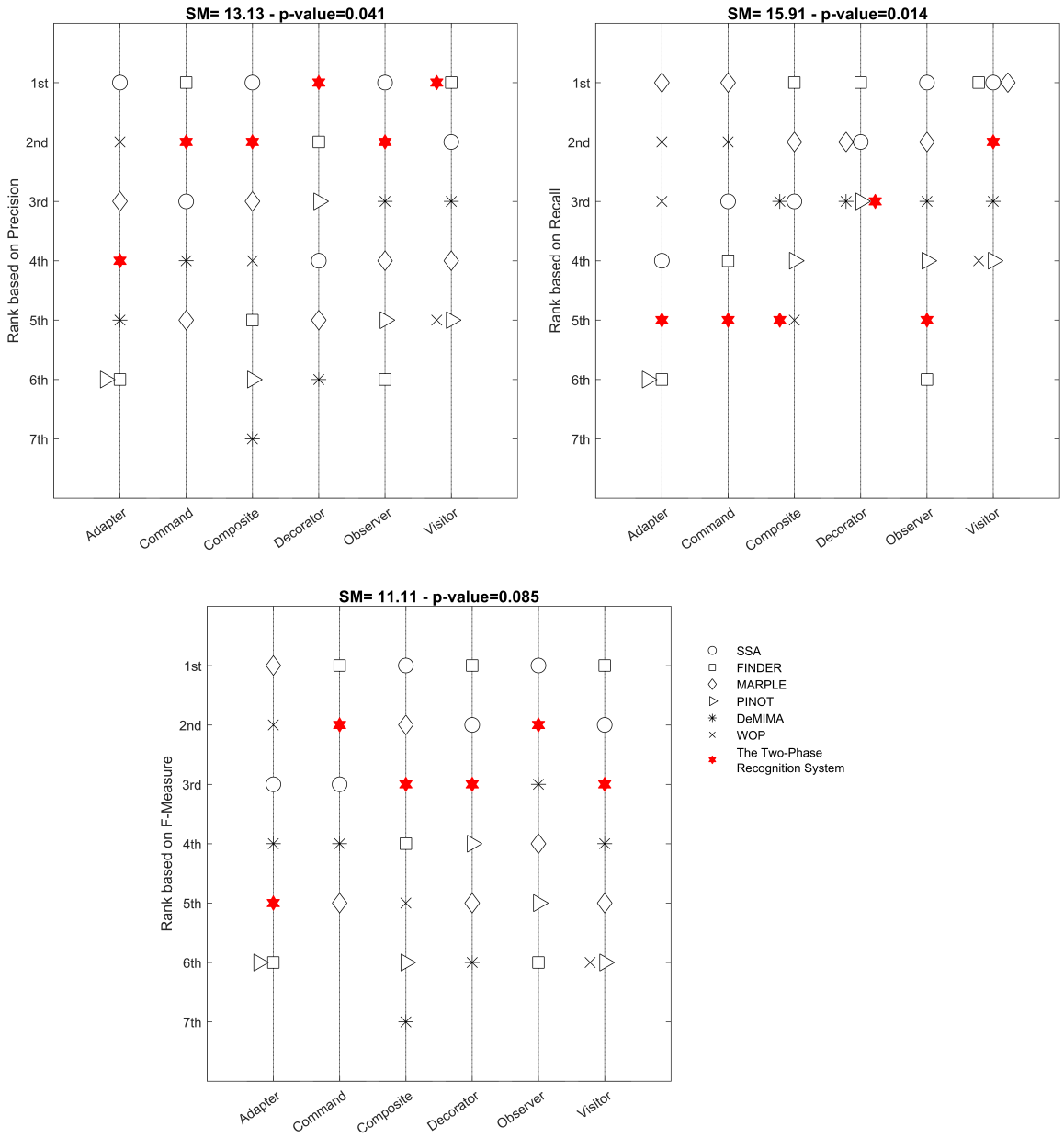



Fig. 6.10 Ranks of the recognition system compared to the other recognition tools based on precision, recall and F-measure.

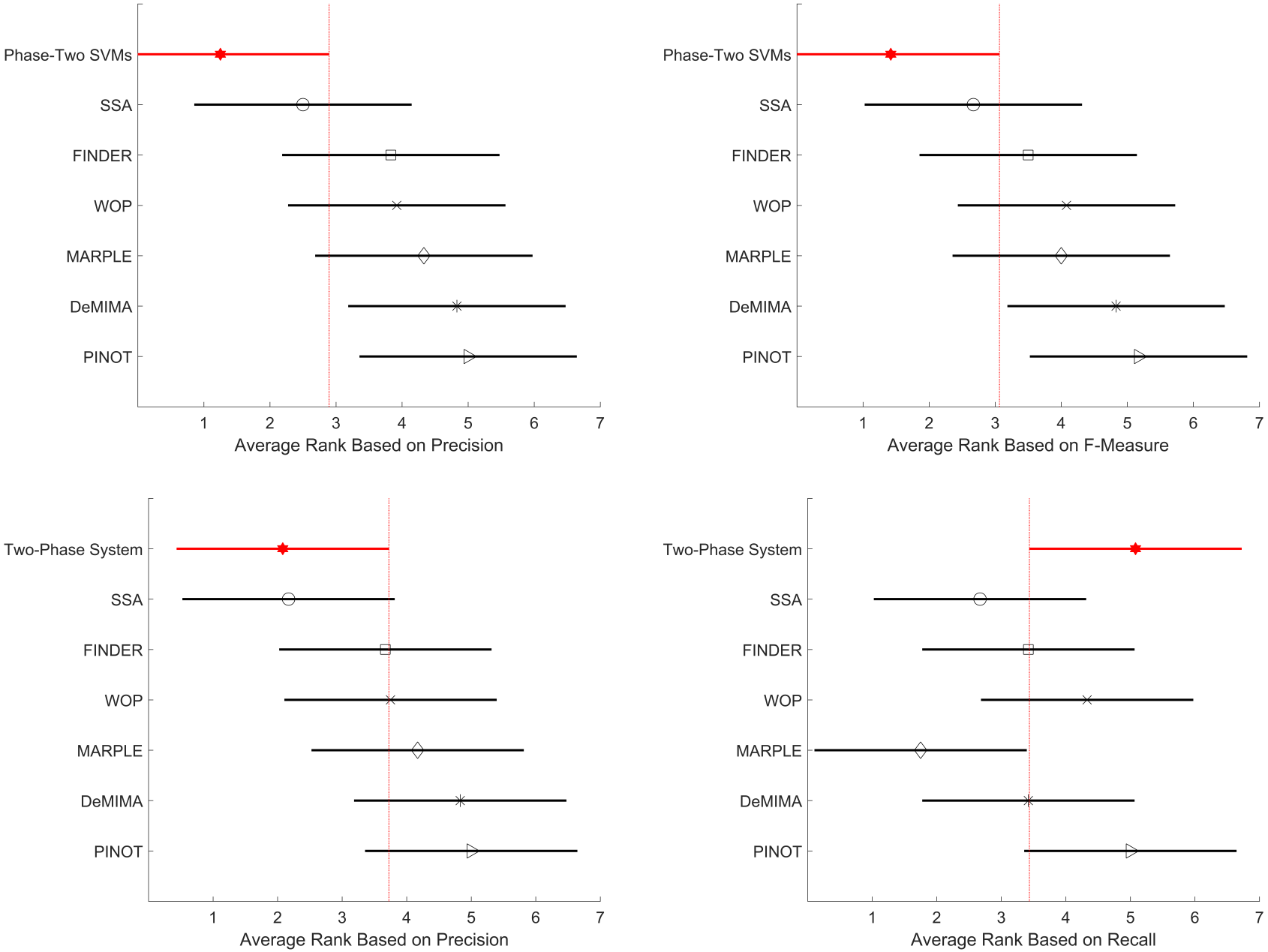



Fig. 6.11 The results of the Bonferroni-Dunn test for the phase-two classifiers (top) and the recognition system (below).

where $q_\alpha$ is the critical value, $k$ is the number of classifiers and $N$ is the number of (DP) datasets. The critical value used is 2.638, which is provided in [48] for comparing seven classifiers at $\alpha = 0.05$ level of significance. This critical value is adjusted for making $k - 1$ comparisons. The results of the post-hoc test are shown in Fig. 6.11, in which it can be seen that the performance of phase-two classifiers, as measured by the precision and the F-Measure, is significantly better than the performance of the DeMIMA and PINOT tools. Although the precision of the two-phase system as a whole is not shown to be significantly different, the results of the SM test shown earlier indicates that such a difference exists but the post-hoc test fails to detect it. The most likely source of the significant difference indicated is, however, the two-phase system as it is not far from being significantly different compared to the DeMIMA and PINOT tools. Nevertheless, the fact that the system still has a leading average of precision ranks, despite the decrease in the number of TP instances, shows that the precision advantage achieved by using the trained classifiers cannot be easily diminished. The performance of the system in terms of its recall, on the other hand, is significantly worse than the MARPLE tool, which has achieved high recall ranks at the cost of recognizing a minimum of about 60 FP instances for 5 of the 6 DPs (and as much as 2398 FP instances in one DP, as mentioned earlier). Also, the significance of this recall difference is caused by phase-one classifiers that have incorrectly filtered out some role-playing classes, which would have otherwise been recognized by phase-two classifiers.

### 6.4.3 Discussion

#### 6.4.3.1 Precision

The precision at which DP instances as well as role-playing classes have been recognized is much better than expected. The fact that role-playing classes, in particular, can actually be fairly accurately recognized is an unforeseen and interesting result.

Although the task of the Component role of the Decorator DP, for example, is essentially to define a set of interface methods, which is a basic and a very common task in OO design, its role-playing classes have been recognized with a perfect recall and a specificity of more than 98%. This level of accuracy is previously thought not to be possible as suggested in [78], in which the authors affirm that role-playing classes cannot be uniquely distinguished but also state that this remains to be empirically validated. Also, although the recognition of Command DP instances, as suggested by the authors of the PINOT tool [150], requires domain-specific knowledge, eight out of its 26 instances have been recognized with a precision of 80% or more. Thus, it can be safely concluded that such high levels of precision have only been enabled by some form of intent modelling, which itself is enabled by information-rich sets of objectively selected features and training datasets of sufficiently high quality.

#### 6.4.3.2 Recall

Although the recall rates are generally not as high as the precision rates, there are only few positive instances in the test set and, as discussed earlier, this does not enable an accurate recall assessment. However, even if the recall performance of the trained classifiers is as the current results suggest, it is not different from the other tools while at the same time they have a significantly better precision. One can then be more confident that recognised instances are indeed true positive ones. The recall performance is, however, more of an issue in phase-one classifiers given that most of them have not achieved a perfect recall rate as hoped for, and this problem manifest itself when the recognition system is tested as a whole. Yet, even in this case, the advantage of the other tools in recall is gained at the cost of recognizing many FP instances, examples of which are discussed in the results.

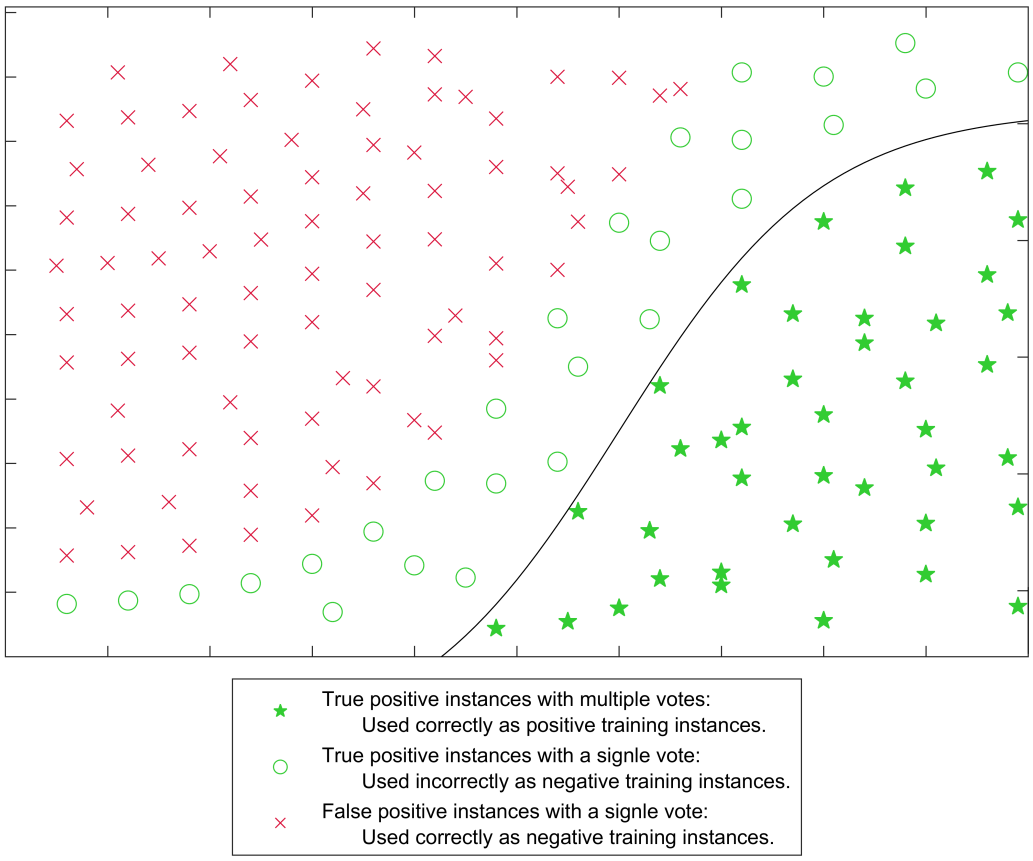



Fig. 6.12 A potential explanation for the relatively low recall rates.

One potential explanation for not having as high recall rates as hoped for is that some of the negative training instances may in fact be positive ones. In other words, some of the true positive instances may have been incorrectly labelled as negative instances (because they have received only a single vote, as explained in Chapter 5). Such mislabelled training instances are likely to be between the correctly labelled positive and negative training instances, as shown in Fig. 6.12. The classification decision boundary would in this case be drawn inside the region of the positive instances or, in other words, the trained classifiers have been forced to learn sub-concepts of the DP concepts. This results in misclassifying some positive instances as negative ones, and hence the relatively low recall rates. This also explains the fact that most negative instances have an estimated probability of less than 1%, as

the results of phase-one classifiers show, given the large distance between them and the classification boundary. This problem can be solved by identifying mislabelled training instances and then remove or relabel them by using the methods suggested in [31] or [106], for example.

A careful and detailed investigation of the role-playing classes and DP instances that are consistently missed by all the classifiers produced by all configurations may also provide another explanation as to why they have been missed. This can actually guide future extension of the feature sets, if any is needed, in order to recover any design information that may not otherwise be recoverable.

#### 6.4.3.3 Context-Based Normalisation and Quality Features

The impact of the context-based normalisation on the recognition accuracy is similar to that of including the quality features: they both are more likely to improve the precision but degrade the recall (Fig. 6.7). In the case of the context-based normalisation, and using the example of *updating* methods provided in Section 3.4.1, it seems that while the normalisation does expose many potential false positive classes by showing that only a small fraction of their methods actually update super class methods, it also makes some true role-playing classes do not appear to be so when they have relatively small fractions of *updating* methods. If, on the other hand, the absolute feature values are used, all classes with a large enough number of *updating* methods will be recognized as role-playing classes, regardless of how many other methods there are. While this set of recognized role-playing classes is likely to include many false positive classes, it also has a better chance to include more true positive ones. Although this is a simplified one-dimensional example used to explain the results of complex multi-dimensional models, it does provide a plausible and logical explanation.

With regards to the effect of including quality features on recognition accuracy, it seems that including them have led to adding some restrictions on the quality characteristics of classes. While these added restrictions have helped to identify and filter out more false positive classes, which explains the improvements in specificity, other true role-playing classes can be filtered out as well if they exhibit different quality characteristics than those expected. It should probably be expected that, even if the role-playing classes of a DP role do generally have a certain quality characteristic, it is not necessarily the case in every single class. So, the 38% of the times (Fig. 6.7) in which the inclusion of quality features have led to a lower recall rate may have been the results of missing such characteristically different classes.

The answers to the questions of whether or not feature values should be contextually normalized and whether or not the quality features should be included depend on the purpose of the task at hand and differ for individual DPs and DP roles. The precision may be more important in one task and the opposite may be true in another. So, if the classifiers trained in this thesis are put together to implement a recognition tool, users will be enabled to answer the aforementioned questions in a configuration window, and the recognition process will be performed by using the classifiers that correspond to the selected configurations.

#### 6.4.3.4 Votes in Dataset Construction

One obvious finding from the results is that the quality of the training datasets are not only determined by the minimum number of votes set for positive examples, but also by the accuracy of the voting tools. This is demonstrated clearly in the result of the Adapter, Visitor and the Proxy DPs, as discussed earlier. However, the minimum number of votes used for the Adapter DP can, and probably should, be increased to compensate for the low accuracy at which this DP is recognised by the voting tools. It should, however, be noted that the trained Adapter DP classifiers

have performed relatively better than the voting tools, but their performance is still not as good as it needs to be. The same cannot be done with the Proxy DP because of the number of its actual voters and the availability of only 10 positive instances if the minimum is increased by one (see Table 5.2).

#### 6.4.3.5 Implication for the Results of DP Quality Impact

The datasets constructed for DP roles in Chapter 5 are used to both produce training datasets for their classifiers as well as to evaluate their impact on software quality. So, the accuracy at which the trained classifiers have performed can be used to either support or question the results of the quality impact evaluation as reported in Section 6.2. Since all DP roles, with the exception of only four, have been recognized with a G-mean of more than 70%, it can be said that this is an evidence supporting the results of the quality impact evaluation. Also, the effect of including quality features on the recognition accuracy can probably be seen as a further supporting evidence. As the pie charts in Fig. 6.7 show, the specificity of roles classifiers, when the quality features are included, is better or at least the same in 81% of the times. The same also applies to the recall, though in only 62% of the times. On the other hand, however, if the potential explanation provided above for the overall relatively low recall rates is found to be true, it may cast some doubt on the results of the quality impact evaluation. This is because it means that the datasets, based on which the evaluation is performed, include negative examples that are actually positive ones. Nevertheless, the results reported for the quality impact of at least the Visitor DP roles as well as the Component role of the Decorator DP are certainly validated given the almost perfect recall and specificity of their classifiers.

## 6.5 Conclusions

A DP recognition system was built in this chapter to evaluate and demonstrate the adequacy of the feature and data sets introduced in previous chapters. The system is implemented following a two-phase approach and considers only two roles for each DP, and the rationale and justification for these decisions have been put forward in this chapter. The problem of representing DP instances as training instances has also been discussed and a new training instance structure has been proposed and used. Feature subsets were then objectively selected for DPs and DP roles, and the quality impact of the latter was also evaluated. Multiple classifier versions were then trained for each DP and DP role, which were objectively evaluated and compared with each other as well as with other recognition tools based on a peer-reviewed benchmark.

The results show that the trained classifiers can recognise role-playing classes and DP instances with a remarkably high precision. The precision levels achieved by the classifiers of some DP roles, in particular, was an unforeseen and interesting result that was previously thought not to be possible. It was also found that the precision is better, in most cases, when the quality metrics are added to the input feature vectors of role classifiers. DP instances have also been recognised by the trained classifiers with a statistically significantly better precision than other DP recognition tools. Although the recall performance of the classifiers is generally not as good as their precision, potential explanations and solutions have been suggested to improve it.

Overall, the accuracy at which instances of the DPs and DP roles have been recognised shows that their *intent* have been captured and modelled by the trained classifiers. It also shows that the feature sets introduced in Chapter 3 and Chapter 4 do provide the information required by the recognition process to succeed, and that the datasets constructed in Chapter 5 are of sufficiently high quality.

# Chapter 7

# Conclusions and Future Work

This thesis has laid out the foundation for a completely new paradigm of research in the field of DP recognition. It has also put forward arguments against existing approaches which rely, to different extents, on subjective recognition rules and have a limited capability to tackle the recognition problem. The thesis has, instead, encouraged and enabled a paradigm shift in the way this problem should be approached, leading to better objective data-driven solutions capable of tackling the problem as it is instead of redefining it into an easier problem. This long overdue paradigm shift has been enabled in the thesis by the novel sets of features introduced as well as the novel approach proposed for DP dataset construction. A new structure for the representation of DP training instances has also been proposed in this thesis.

Although the results reported in this thesis are generally good and encouraging, the thesis does not claim to have solved the problem of recognising the set of DPs included in the experiments. Instead, the results reported are seen as a step in the right direction for a more effective, accurate and efficient solutions for the problem of DP recognition. It is also hoped that the thesis will help in improving the standards of research in this field, particularly in how recognition approaches

should be evaluated and compared. This is because no real progress can be made without objective evaluation of new approaches.

The following sections three sections outline the main contributions of this thesis. Then, in Section 7.4, some directions and opportunities for future research are provided.

## 7.1 Intent Reasoning

The main advantage of the recognition approach proposed in this thesis, as shown by the experiment results, is the precision at which DP instances have been recognised. It can be safely assumed that such levels of precision could not be achieved without modelling the intent of the DPs to be recognised. After all, the difference between the positive and negative training examples is basically in their intents, since they are otherwise similar in their tangible characteristics given that they all are recognised by the same set of recognition tools. This is perfectly demonstrated by the example given in Section 6.4.3 which shows how well classes playing the Component role in Decorator DP instances have been recognised. Recognizing such role-playing classes that have a very common set of characteristics at such a high level of accuracy undoubtedly requires reasoning about their intent. The fact that this high level of accuracy is even possible, not only in the recognition of DP instances but also in the recognition of classes playing DP roles, which have less known characteristics by which they can be distinguished, is a practical validation of the used feature sets and training datasets as well as the approach proposed to construct the latter.

## 7.2 Feature Sets and Quality Impact

The set of features introduced in this thesis covers a wide range of design aspects and properties and includes many novel ones. A set of novel structural features were proposed to recover information about class associations accurately and unambiguously. To recover more information about how different classes interact, a novel set of 64 behavioural features were proposed. Besides the default (i.e. absolute) method to calculate the value of these features, a new alternative calculation method (i.e. context-based normalization) has also been proposed, which turns out to have a positive effect on the recognition precision.

The proposed novel quality metrics, on the other hand, were proposed to measure and capture the impact on coupling aspects that are relevant to DPs. A novel threshold-based normalization approach has also been introduced to account for the overall number of classes in software systems when calculating the values of the coupling metrics. The impact of DP on software quality, as measured by the coupling metrics as well as the cohesion and complexity metrics, was analysed. The importance of this impact analysis study is that it is based on the largest and most accurate DP dataset, and it helps to fill the identified gap of evaluating the impact of DPs based on OO metrics. The ultimate aim of the study is, however, to enrich the recognition process with more features, which have been found to improve its precision.

The set of features, as a whole, is a novel one although it does include some existing metrics. The novelty comes from putting together features that capture and collect as much information as possible about all the identified properties and aspects of software design. This set can be seen as a grand global set of features, which can be used as a single global set or multiple global sets designed to suit different phases of the recognition process, as it was the case in this thesis.

## 7.3 Construction of Training Datasets

One of the main obstacles to employing data-driven and machine learning based approaches is the lack of a large and reasonably accurate DP dataset and the challenges of constructing one. In order to overcome this obstacle, a novel and fully automated approach was proposed and implemented in this thesis to construct 21 datasets (i.e. for the 7 DPs and their 14 roles) based on 539 software systems and by using 6 DP recognition tools. This set of constructed DP datasets is the largest and most accurate one in existence. Their accuracy stems from the fact that the approach used to produce them was designed in a way that does not only improve the accuracy of the added positive examples, but also the accuracy of the negative ones. These datasets formed a fundamental element that has enabled the work on this thesis to be carried out, and they will be made publicly available for interested researchers. This set of datasets can be easily further improved in terms of accuracy and size by analysing the 539 systems by using more tools, which enables increasing the minimum number of votes required for instances to be used as positive examples. Also, it can be improved by using the same set of tools to analyse more systems, which will add more examples to the datasets.

## 7.4 Future Work

Given the foundation laid out by this thesis for a new paradigm of research in DP recognition, there are seemingly countless opportunities for improvements, methods to be used and directions to take. The following, however, presents the three main fronts of future research:

- The ongoing work on pattern recognition and machine learning has produced a wealth of algorithms, methods and models that have helped to solve complex real-world problems, and all of this wealth has now been made possible to

use in DP recognition research. There are, for example, many methods and techniques to preprocess the training dataset, including outlier detection techniques, feature selection and oversampling methods. Although such methods have been used in this thesis, other methods may lead to better training dataset and, consequently, better accuracy performance. For example, using a different oversampling method, or even a different approach (e.g. using cost-sensitive methods) to deal with the problem of learning from imbalanced datasets, may improve on the results obtained in this thesis. There are, on the other hand, many machine learning models and algorithms (e.g. back-propagation artificial neural networks) which may also lead to better accuracy performance.

- The datasets themselves can be improved in terms of their accuracy and size by plugging in more tools and/or software systems, as mentioned above. However, the accuracy can be improved with the datasets as they are now by changing the minimum number of votes set for the positive examples, one example of which is the increase suggested for the Adapter DP in Section 6.4.3. Having trained and evaluated several types of machine learning models, they can themselves be used recursively as voters in the construction of the datasets, which can lead to an ever improving dataset accuracy. More datasets can also be constructed for other DPs following the approach proposed in this thesis. Although, theoretically, classifiers trained by a dataset constructed from Java software systems, for example, can be used to recognise DP instances on systems implemented by using other programming languages, they may not perform as well given the differences between languages in metric value distributions. So, it may be more appropriate to train the classifiers by using training examples from the same type of systems in which the DPs are to be

recognised, and the approach proposed in this thesis provides a feasible way to do that.

- The set of features introduced in this thesis can also be improved by adding more features to recover more and more design information. Investigating the characteristics of instances that are consistently missed by the classifiers may provide a helpful guide in this direction. This is, in fact, one of the solutions suggested in Section 6.4.3 to improve the recall performance.

# References


[1] Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Constructions*. Center for Environmental Structure Berkeley, Calif: Center for Environmental Structure series. Oxford University Press.

[2] Alhusain, S., Coupland, S., John, R., and Kavanagh, M. (2013a). Design pattern recognition by using adaptive neuro fuzzy inference system. In *Proceedings of the IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 581–587.

[3] Alhusain, S., Coupland, S., John, R., and Kavanagh, M. (2013b). Towards machine learning based design pattern recognition. In *Proceedings of the 13th UK Workshop on Computational Intelligence (UKCI)*, pages 244–251.

[4] Alpaydin, E. (2010). *Introduction to Machine Learning (Adaptive Computation and Machine Learning series)*. The MIT Press, 2nd edition.

[5] Alur, D., Malks, D., and Crupi, J. (2001). *Core J2EE Patterns: Best Practices and Design Strategies (2nd Edition)*. Prentice Hall.

[6] Alves, T., Ypma, C., and Visser, J. (2010). Deriving metric thresholds from benchmark data. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 1–10.

[7] Ampatzoglou, A., Charalampidou, S., and Stamelos, I. (2013). Research state of the art on GoF design patterns: A mapping study. *Journal of Systems and Software*, 86(7).

[8] Ampatzoglou, A. and Chatzigeorgiou, A. (2007). Evaluation of object-oriented design patterns in game development. *Information and Software Technology*, 49(5):445 – 454.

[9] Ampatzoglou, A., Chatzigeorgiou, A., Charalampidou, S., and Avgeriou, P. (2015). The effect of GoF design patterns on stability: A case study. *IEEE Transactions on Software Engineering*, 41(8):781–802.

[10] Ampatzoglou, A., Michou, O., and Stamelos, I. (2012). Building and mining a repository of design pattern instances: Practical and research benefits. *Entertainment Computing*.

[11] Antoniol, G., Casazza, G., Di Penta, M., and Fiutem, R. (2001). Object-oriented design patterns recovery. *Journal of Systems and Software*, 59(2):181–196.

[12] Arauzo-Azofra, A., Aznarte, J. L., and Benítez, J. M. (2011). Empirical study of feature selection methods based on individual feature evaluation for classification problems. *Expert Systems with Applications*, 38(7):8170–8177.

[13] Arcelli, F. and Cristina, L. (2007). Enhancing software evolution through design pattern detection. In *Proceedings of The Third International IEEE Workshop on Software Evolvability*, pages 7–14.

[14] Arcelli, F., Zanoni, M., and Caracciolo, A. (2010). A benchmark platform for design pattern detection. In *Proceedings of the 2nd International Conference on Pervasive Patterns and Applications (PATTERNS)*.

[15] Arcelli Fontana, F., Maggioni, S., and Raibulet, C. (2011). Understanding the relevance of micro-structures for design patterns detection. *Journal of Systems and Software*, 84(12):2334–2347.

[16] Arcelli Fontana, F. and Zanoni, M. (2011). A tool for design pattern detection and software architecture reconstruction. *Information sciences*, 181(7):1306–1324.

[17] Balanyi, Z. and Ferenc, R. (2003). Mining design patterns from c++ source code. In *Proceedings of the International Conference on Software Maintenance*, pages 305–314.

[18] Bansiya, J. and Davis, C. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17.

[19] Batuwita, R. and Palade, V. (2013). *Class Imbalance Learning Methods for Support Vector Machines*, pages 83–99. John Wiley and Sons, Inc.

[20] Bender, R. (1999). Quantitative risk assessment in epidemiological studies investigating threshold effects. *Biometrical Journal*, 41(3):305–319.

[21] Benlarbi, S., El Emam, K., Goel, N., and Rai, S. (2000). Thresholds for object-oriented measures. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 24–38.

[22] Bernardi, M. L., Cimitile, M., and Di Lucca, G. (2014). Design pattern detection using a dsl-driven graph matching approach. *Journal of Software: Evolution and Process*, 26(12):1233–1266.

[23] Bieman, J., Straw, G., Wang, H., Munger, P., and Alexander, R. (2003). Design patterns and change proneness: an examination of five evolving systems. In *The Ninth International Software Metrics Symposium*, pages 40–49.

[24] Binun, A. (2012). *High Accuracy Design Pattern Detection*. PhD thesis, Universitäts-und Landesbibliothek Bonn.

[25] Binun, A. and Kniesel, G. (2012). Dpjf - design pattern detection with high accuracy. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 245–254.

[26] Bjork, S. and Holopainen, J. (2004). *Patterns in Game Design*. Game Development Series. Charles River Media.

[27] Briand, L., Daly, J., and Wust, J. (1998). A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117.

[28] Briand, L., Daly, J., and Wust, J. (1999a). A unified framework for coupling measurement in object-oriented systems. *Software Engineering, IEEE Transactions on*, 25(1):91–121.

[29] Briand, L., Devanbu, P., and Melo, W. (1997). An investigation into coupling measures for c++. In *Proceedings of the 19th International Conference on Software Engineering*, ICSE '97, pages 412–421, New York, NY, USA. ACM.

[30] Briand, L., Wust, J., and Lounis, H. (1999b). Using coupling measurement for impact analysis in object-oriented systems. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 475–482.

[31] Brodley, C. E. and Friedl, M. A. (1999). Identifying mislabeled training data. *Journal of Artificial Intelligence Research*, pages 131–167.

[32] Brown, G., Pocock, A., Zhao, M.-J., and Luján, M. (2012). Conditional likelihood maximisation: A unifying framework for information theoretic feature selection. *J. Mach. Learn. Res.*, 13(1):27–66.

[33] Brown, W., Malveau, R., McCormick, H. W. S., and Mowbray, T. J. (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley.

[34] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., and Stal, M. (1996). *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. John Wiley & Sons.

[35] Campbell, C. and Ying, Y. (2011). *Learning with Support Vector Machines (Synthesis Lectures on Artificial Intelligence and Machine Learning)*. Morgan & Claypool Publishers.

[36] Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27.

[37] Chatfield, M. and Mander, A. (2009). The skillings–mack test (friedman test when there are missing data). *The Stata Journal*, 9(2):299.

[38] Chidamber, S. and Kemerer, C. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

[39] Chihada, A., Jalili, S., Hasheminejad, S. M. H., and Zangooei, M. H. (2015). Source code and design conformance, design pattern detection from source code by classification approach. *Applied Soft Computing*, 26(0):357 – 367.

[40] Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273–297.

[41] Costagliola, G., De Lucia, A., Deufemia, V., Gravino, C., and Risi, M. (2006). Case studies of visual language based design patterns recovery. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10 pp.–174.

[42] Czibula, I. and Czibula, G. (2008). Identifying design patterns in object-oriented software systems using unsupervised learning. In *Automation, Quality and Testing, Robotics, 2008. AQTR 2008. IEEE International Conference on*, volume 3, pages 347–352.

[43] Dabain, H. (2011). A graphical notation for design pattern detection. Master's thesis, Graduate Program in Computer Science and Engineering, York University, Toronto, Canada.

[44] Dabain, H., Manzer, A., and Tzerpos, V. (2015). Design pattern detection using FINDER. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1586–1593, New York, NY, USA.

[45] Dallal, J. A. (2013). Object-oriented class maintainability prediction using internal quality attributes. *Information and Software Technology*, 55(11):2028 – 2048.

[46] D'Ambros, M., Lanza, M., and Robbes, R. (2010). An extensive comparison of bug prediction approaches. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, pages 31 – 41. IEEE CS Press.

[47] Dash, M. and Liu, H. (1997). Feature selection for classification. *Intelligent data analysis*, 1(1-4):131–156.

[48] Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30.

[49] Di Penta, M., Cerulo, L., Guéhéneuc, Y., and Antoniol, G. (2008). An empirical study of the relationships between design pattern roles and class change proneness. In *IEEE International Conference on Software Maintenance*, pages 217–226.

[50] Dietrich, J. and Elgar, C. (2007). Towards a web of patterns. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):108–116.

[51] Dong, J., Sun, Y., and Zhao, Y. (2008). Compound record clustering algorithm for design pattern detection by decision tree learning. In *Information Reuse and Integration, 2008. IRI 2008. IEEE International Conference on*, pages 226–231.

[52] Dong, J., Zhao, Y., and Peng, T. (2009a). A review of design pattern mining techniques. *International Journal of Software Engineering and Knowledge Engineering*, 19(6):823–855.

[53] Dong, J., Zhao, Y., and Sun, Y. (2009b). A matrix-based approach to recovering design patterns. *IEEE Transactions on Systems, Man and Cybernetics*, 39(6):1271–1282.

[54] Dreyfus, G. and Guyon, I. (2006). Assessment methods. In Guyon, I., Nikravesh, M., Gunn, S., and Zadeh, L. A., editors, *Feature Extraction: Foundations and Applications*, pages 65–88. Springer Berlin Heidelberg, Berlin, Heidelberg.

[55] Dubey, S. K. and Rana, A. (2011). Assessment of maintainability metrics for object-oriented software system. *SIGSOFT Softw. Eng. Notes*, 36(5):1–7.

[56] Dubois, D. and Prade, H. (2001). Possibility theory, probability theory and multiple-valued logics: A clarification. *Annals of mathematics and Artificial Intelligence*, 32(1-4):35–66.

[57] Duda, R. O., Hart, P. E., and Stork, D. G. (2001). *Pattern Classification*. Wiley-Interscience.

[58] Eder, J., Kappel, G., and Schrefl, M. (1994). Coupling and cohesion in object-oriented systems. Technical report, University of Klagenfurt.

[59] El-Emam, K. (2002). Object-oriented metrics: A review of theory and practice. In Erdogmus, H. and Tanir, O., editors, *Advances in Software Engineering*, pages 23–50. Springer-Verlag New York, Inc., New York, NY, USA.

[60] Fenton, N. (1990). Software metrics: theory, tools and validation. *Software Engineering Journal*, 5(1):65–78.

[61] Fenton, N. E. and Neil, M. (1999). Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2–3):149 – 157.

[62] Fenton, N. E. and Pfleeger, S. L. (1998). *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition.

[63] Ferenc, R., Beszedes, A., Fulop, L., and Lele, J. (2005). Design pattern mining enhanced by machine learning. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 295–304.

[64] Ferenc, R., Beszedes, A., Tarkiainen, M., and Gyimothy, T. (2002). Columbus - reverse engineering tool and schema for c++. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 172–181.

[65] Ferreira, K. A., Bigonha, M. A., Bigonha, R. S., Mendes, L. F., and Almeida, H. C. (2012). Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2):244 – 257.

[66] Fontana, F. A., Zanoni, M., and Maggioni, S. (2011). Using design pattern clues to improve the precision of design pattern detection tools. *Journal of Object Technology*, 10(4):1–31.

[67] Fowler, M. (1996). *Analysis Patterns: Reusable Object Models*. Addison-Wesley Professional, 1 edition.

[68] Fowler, M. (2004). *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional.

[69] Freeman, E., Bates, B., Sierra, K., and Robson, E. (2004). *Head First Design Patterns*. O'Reilly Media.

[70] Fulop, L., Ferenc, R., and Gyimothy, T. (2008). Towards a benchmark for evaluating design pattern miner tools. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, pages 143–152.

[71] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education.

[72] Gatrell, M. and Counsell, S. (2011). Design patterns and fault-proneness a study of commercial C# software. In *Fifth International Conference on Research Challenges in Information Science*, pages 1–8.

[73] Guéhéneuc, Y.-G. (2007). P-MARt: Pattern-like micro architecture repository. *Proceedings of The 1st EuroPLoP Focus Group on Pattern Repositories*.

[74] Guéhéneuc, Y.-G. and Albin-Amiot, H. (2004). Recovering binary class relationships: Putting icing on the uml cake. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 301–314. ACM.

[75] Guéhéneuc, Y.-G. and Antoniol, G. (2008). DeMIMA: A multilayered approach for design pattern identification. *IEEE Transactions on Software Engineering*, 34(5):667–684.

[76] Guéhéneuc, Y.-G., Guyomarc'h, J.-Y., Khosravi, K., and Sahraoui, H. (2006). Design patterns as laws of quality. In Garzás, J. and Piattini, M., editors, *Object-Oriented Design Knowledge: Principles, Heuristics and Best Practices*. IGI Global, Hershey, USA.

[77] Guéhéneuc, Y.-G., Guyomarch, J.-Y., and Sahraoui, H. (2010). Improving design-pattern identification: a new approach and an exploratory study. *Software Quality Journal*, 18(1):145–174.

[78] Guéhéneuc, Y.-G., Sahraoui, H., and Zaidi, F. (2004). Fingerprinting design patterns. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 172–181.

[79] Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182.

[80] Hahsler, M. (2005). A quantitative study of the adoption of design patterns by open source software developers. In Koch, S., editor, *Free/Open Source Software Development*, chapter V, pages 103–123. Idea Group Publishing.

[81] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18.

[82] Hall, M. A. (1998). *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand.

[83] Han, H., Wang, W.-Y., and Mao, B.-H. (2005). Borderline-SMOTE: A new over-sampling method in imbalanced data sets learning. In Huang, D.-S., Zhang, X.-P., and Huang, G.-B., editors, *Advances in Intelligent Computing: International Conference on Intelligent Computing*, pages 878–887. Springer, Berlin, Heidelberg.

[84] Haqqie, S. and Shahid, A. (2005). Mining design patterns for architecture reconstruction using an expert system. In *9th International Multitopic Conference, IEEE INMIC 2005*, pages 1–6.

[85] He, H., Bai, Y., Garcia, E., and Li, S. (2008). ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pages 1322–1328.

[86] He, H. and Garcia, E. (2009). Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284.

[87] Hegedus, P., Bán, D., Ferenc, R., and Gyimóthy, T. (2012). Myth or reality? analyzing the effect of design patterns on software maintainability. In Kim, T.-h., Ramos, C., Kim, H.-k., Kiumi, A., Mohammed, S., and Ślęzak, D., editors, *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*, volume 340 of *Communications in Computer and Information Science*, pages 138–145. Springer Berlin Heidelberg.

[88] Hensgen, P. (2016). Umbrello uml modeller. https://umbrello.kde.org/.

[89] Hitz, M. and Montazeri, B. (1995). Measuring coupling and cohesion in object-oriented systems. In *Proceedings of International Symposium on Applied Corporate Computing*.

[90] Hsu, C.-W., Chang, C.-C., and Lin, C.-J. (2003). A practical guide to support vector classification. Technical report, Department of Computer Science, National Taiwan University.

[91] Issaoui, I., Bouassida, N., and Ben-Abdallah, H. (2012). A design pattern detection approach based on semantics. In Lee, R., editor, *Software Engineering Research, Management and Applications*, volume 430 of *Studies in Computational Intelligence*, pages 49–63. Springer Berlin Heidelberg.

[92] Izurieta, C. and Bieman, J. (2013). A multiple case study of design pattern decay, grime, and rot in evolving software systems. *Software Quality Journal*, 21(2):289–323.

[93] Jing, L., Keqing, H., Yutao, M., and Rong, P. (2006). Scale free in software metrics. In *Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International*, volume 1, pages 229–235.

[94] Jureczko, M. and Madeyski, L. (2010). Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 9:1–9:10, New York, NY, USA. ACM.

[95] Kaczor, O., Guéhéneuc, Y.-G., and Hamel, S. (2010). Identification of design motifs with pattern matching algorithms. *Information and Software Technology*, 52(2):152 – 168.

[96] Kantardzic, M. (2011). *Data Mining: Concepts, Models, Methods, and Algorithms*. John Wiley & Sons, Inc.

[97] Katsiantis, S., Kanellopoulos, D., and Pintelas, P. (2006). Data preprocessing for supervised learning. *Int. J. Comput. Sci*, 1(2):111–117.

[98] Keerthi, S. S. and Lin, C.-J. (2003). Asymptotic behaviors of support vector machines with gaussian kernel. *Neural computation*, 15(7):1667–1689.

[99] Khomh, F., Guéhéneuc, Y., and Antoniol, G. (2009). Playing roles in design patterns: An empirical descriptive and analytic study. In *IEEE International Conference on Software Maintenance*, pages 83–92.

[100] King, G. and Zeng, L. (2001). Logistic regression in rare events data. *Political Analysis*, 9(2):137–163.

[101] Kitchenham, B., Pfleeger, S., Pickard, L., Jones, P., Hoaglin, D., El Emam, K., and Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *Software Engineering, IEEE Transactions on*, 28(8):721–734.

[102] Kniesel, G. and Binun, A. (2009). Standing on the shoulders of giants - a data fusion approach to design pattern detection. In *Proceedings of the IEEE 17th International Conference on Program Comprehension*, pages 208–217.

[103] Kononenko, I. (1994). Estimating attributes: analysis and extensions of relief. In *Proceedings of the European Conference on Machine Learning*, pages 171–182.

[104] Kuchana, P. (2004). *Software architecture design patterns in Java*. CRC Press.

[105] Kukar, M. and Kononenko, I. (1998). Cost-sensitive learning with neural networks. In *Proceedings of the 13th European Conference on Artificial Intelligence*, pages 445–449.

[106] Lallich, S., Muhlenbach, F., and Zighed, D. A. (2002). Improving classification by removing or relabeling mislabeled instances. In *Foundations of Intelligent Systems*, pages 5–15. Springer.

[107] Lanza, M., Ducasse, S., Gall, H., and Pinzger, M. (2005). Codecrawler - an information visualization tool for program comprehension. In *Proceedings of the 27th International Conference on Software Engineering*, pages 672–673.

[108] Lanza, M. and Marinescu, R. (2006). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer.

[109] Li, W. and Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122.

[110] Lin, H.-T., Lin, C.-J., and Weng, R. C. (2007). A note on platt's probabilistic outputs for support vector machines. *Machine Learning*, 68(3):267–276.

[111] Lindsay, J., Noble, J., and Tempero, E. (2010). Does size matter?: A preliminary investigation of the consequences of powerlaws in software. In *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, WETSoM '10, pages 16–23, New York, NY, USA. ACM.

[112] Liu, H. and Setiono, R. (1996). A probabilistic approach to feature selection - a filter solution. In *13th International Conference on Machine Learning*, pages 319–327.

[113] Louridas, P., Spinellis, D., and Vlachos, V. (2008). Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18(1):2:1–2:26.

[114] Lu, H., Zhou, Y., Xu, B., Leung, H., and Chen, L. (2012). The ability of object-oriented metrics to predict change-proneness: a meta-analysis. *Empirical Software Engineering*, 17(3):200–242.

[115] Lucia, A. D., Deufemia, V., Gravino, C., and Risi, M. (2009). Design pattern recovery through visual language parsing and source code analysis. *Journal of Systems and Software*, 82(7):1177–1193.

[116] McC. Smith, J. (2002). An elemental design pattern catalog. Technical Report TR02-040, University of North Carolina.

[117] McCabe, T. (1976). A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320.

[118] Menzies, T., Krishna, R., and Pryor, D. (2015). The promise repository of empirical software engineering data. http://openscience.us/repo. North Carolina State University, Department of Computer Science.

[119] Mojzes, M., Rost, M., Smolka, J., and Virius, M. (2014). Feature space for statistical classification of java source code patterns. In *Control Conference (ICCC), 2014 15th International Carpathian*, pages 357–361.

[120] Morasca, S. (2009). A probability-based approach for measuring external attributes of software artifacts. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 44–55, Washington, DC, USA. IEEE Computer Society.

[121] Negnevitsky, M. (2002). *Artificial Intelligence: A Guide to Intelligent Systems*. Addison-Wesley Longman.

[122] Nickel, U., Niere, J., and Zündorf, A. (2000). The fujaba environment. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 742–745, Limerick, Ireland.

[123] Niere, J. (2002). Fuzzy logic based interactive recovery of software design. In *Proceedings of the 24rd International Conference on Software Engineering*, pages 727–728.

[124] Niere, J., Meyer, M., and Wendehals, L. (2004). User-driven adaption in rule-based pattern recognition. Technical Report tr-ri-04-249, University of Paderborn.

[125] Niere, J., Schafer, W., Wadsack, J., Wendehals, L., and Welsh, J. (2002). Towards pattern-based design recovery. In *Proceedings of the 24rd International Conference on Software Engineering*, pages 338–348.

[126] Niere, J., Wadsack, J., and Wendehals, L. (2003). Handling large search space in pattern-based reverse engineering. In *Proceedings of The 11th IEEE International Workshop on Program Comprehension*, pages 274–279.

[127] Olague, H. M., Etzkorn, L. H., Messimer, S. L., and Delugach, H. S. (2008). An empirical validation of object-oriented class complexity metrics and their ability

to predict error-prone classes in highly iterative, or agile, software: a case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(3):171–197.

[128] Oliveira, P., Valente, M. T., and Lima, F. P. (2014). Extracting relative thresholds for source code metrics. In *Proceedings of the Working Conference on Reverse Engineering (WCRE) and the European Conference on Software Maintenance and Reengineering (CSMR)*.

[129] Parsons, D. (2012). Objects working together: Association, aggregation, and composition. In *Foundational Java*, pages 125–157. Springer London.

[130] Peng, H., Long, F., and Ding, C. (2005). Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(8):1226–1238.

[131] Peters, F., Menzies, T., and Marcus, A. (2013). Better cross company defect prediction. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 409–418.

[132] Pettersson, N., Lowe, W., and Nivre, J. (2010). Evaluation of accuracy in design pattern occurrence detection. *IEEE Transactions on Software Engineering*, 36(4):575–590.

[133] Philippow, I., Streitferdt, D., Riebisch, M., and Naumann, S. (2005). An approach for reverse engineering of design patterns. *Software & Systems Modeling*, 4(1):55–70.

[134] Prechelt, L., Unger, B., Tichy, W., Brossler, P., and Votta, L. (2001). A controlled experiment in maintenance: comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 27(12):1134–1144.

[135] Prechelt, L., Unger-Lamprecht, B., Philippsen, M., and Tichy, W. (2002). Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *Software Engineering, IEEE Transactions on*, 28(6):595–606.

[136] Radjenovic, D., Herico, M., Torkar, R., and Zivkovic, A. (2013). Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55:1397–1418.

[137] Rasool, G. and Mäder, P. (2014). A customizable approach to design patterns recognition based on feature types. *Arabian Journal for Science and Engineering*, 39(12):8851–8873.

[138] Riaz, M., Breaux, T., and Williams, L. (2015). How have we evaluated software pattern application? a systematic mapping study of research design practices. *Information and Software Technology*, 65:14–38.

[139] Rodriguez, D., Herraiz, I., and Harrison, R. (2012). On software engineering repositories and their open problems. In *Proceedings of the First International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, pages 52–56.

[140] Romano, S., Scanniello, G., Risi, M., and Gravino, C. (2011). Clustering and lexical information support for the recovery of design pattern in source code. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 500–503.

[141] Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling Language Reference Manual*. ADDISON WESLEY.

[142] Saksena, M., France, R., and Larrondo-Petrie, M. (1998). A characterization of aggregation. In Rolland, C. and Grosz, G., editors, *OOIS'98*, pages 11–19. Springer London.

[143] Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F. (2000). *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons.

[144] Scientific Toolworks, Inc. (2016). Understand™ static code analysis tool. https://scitools.com/.

[145] Sellers, B. H. (1996). *Ojbect-Oriented Metrics. Measures of Complexity*. Prentice Hall.

[146] Shatnawi, R. (2010). A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. *IEEE Transactions on Software Engineering*, 36(2):216–225.

[147] Shatnawi, R. and Althebyan, Q. (2013). An empirical study of the effect of power law distribution on the interpretation of oo metrics. *ISRN Software Engineering*, 2013.

[148] Shatnawi, R., Li, W., Swain, J., and Newman, T. (2010). Finding software metrics threshold values using ROC curves. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(1):1–16.

[149] Shepperd, M., Chidamber, S., and Kemerer, C. (1995). Comments on "a metrics suite for object oriented design". *Software Engineering, IEEE Transactions on*, 21(3):263–265.

[150] Shi, N. and Olsson, R. (2006). Reverse engineering of design patterns from java source code. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 123–134.

[151] Shin, K.-S., Lee, T. S., and jung Kim, H. (2005). An application of support vector machines in bankruptcy prediction model. *Expert Systems with Applications*, 28(1):127 – 135.

[152] Sommerville, I. (2011). *Software Engineering*. Addison-Wesley, 9 edition.

[153] Stevens, W., Myers, G., and Constantine, L. (1974). Structured design. *IBM Systems Journal*, 13(2):115–139.

[154] Taibi, T. (2007). *Design Pattern Formalization Techniques*. IGI Publishing.

[155] Taube-Schock, C., Walker, R., and Witten, I. (2011). Can we avoid high coupling? In Mezini, M., editor, *ECOOP 2011 – Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 204–228. Springer Berlin Heidelberg.

[156] Tegarden, D. P., Sheetz, S. D., and Monarchi, D. E. (1995). A software complexity model of object-oriented systems. *Decision Support Systems*, 13(3–4):241 – 262.

[157] Tempero, E., Counsell, S., and Noble, J. (2010). An empirical study of overriding in open source java. In *Proceedings of the 33rd Australasian Conferenc on Computer Science*, ACSC '10, pages 3–12. Australian Computer Society, Inc.

[158] Tonella, P. and Antoniol, G. (2001). Inference of object-oriented design patterns. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(5):309–330.

[159] Torkkola, K. (2006). Information-theoretic methods. In Guyon, I., Nikravesh, M., Gunn, S., and Zadeh, L., editors, *Feature Extraction*, volume 207 of *Studies in Fuzziness and Soft Computing*, pages 167–185. Springer Berlin Heidelberg.

[160] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S. (2006). Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909.

[161] Uchiyama, S., Kubo, A., Washizaki, H., and Fukazawa, Y. (2014). Detecting design patterns in object-oriented program source code by using metrics and machine learning. *Journal of Software Engineering and Applications*, 7(12):983–998.

[162] Visual Paradigm (2016). Visual paradigm for UML. https://www.visual-paradigm.com/.

[163] Vokac, M. (2004). Defect frequency and design patterns: an empirical study of industrial code. *IEEE Transactions on Software Engineering*, 30(12):904–917.

[164] Wang, Y., Guo, H., Liu, H., and Abraham, A. (2012). A fuzzy matching approach for design pattern mining. *Journal of Intelligent and Fuzzy Systems*, 23(2):53–60.

[165] Wenz, J. (2015). Why NASA needs a programmer fluent in 60 year old languages. http://www.popularmechanics.com/space/a17991/voyager-1-voyager-2-retiring-engineer/. Accessed: 2016-02-07.

[166] Weston, J., Mukherjee, S., Chapelle, O., Pontil, M., Poggio, T., and Vapnik, V. (2001). Feature selection for SVMs. In Leen, T., Dietterich, T., and Tresp, V., editors, *Advances in Neural Information Processing Systems 13*, pages 668–674. MIT Press.

[167] Witten, I. H., Frank, E., and Hall, M. A. (2011). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 3 edition.

[168] Xenos, M., Stavrinoudis, D., Zikouli, K., and Christodoulakis, D. (2000). Object-oriented metrics: a survey. In *Proceedings of the Federation of European Software Measurement Associations*, pages 1–10.

[169] Yang, S., Manzer, A., and Tzerpos, V. (2015). Measuring the quality of design pattern detection results. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 53–62.

[170] Yu, D., Zhang, Y., and Chen, Z. (2015). A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. *Journal of Systems and Software*, 103(0):1 – 16.

[171] Yu, H., Yang, J., Han, J., and Li, X. (2005). Making SVMs scalable to large data sets using hierarchical cluster indexing. *Data Mining and Knowledge Discovery*, 11(3):295–321.

[172] Zanoni, M. (2012). *Data mining techniques for design pattern detection*. PhD thesis, Università degli Studi di Milano-Bicocca.

[173] Zanoni, M., Fontana, F. A., and Stella, F. (2015). On applying machine learning techniques for design pattern detection. *Journal of Systems and Software*, 103:102 – 117.

[174] Zelkowitz, M. and Wallace, D. (1998). Experimental models for validating technology. *Computer*, 31(5):23–31.

[175] Zhang, C. and Budgen, D. (2012). What do we know about the effectiveness of software design patterns? *Software Engineering, IEEE Transactions on*, 38(5):1213–1231.

[176] Zhang, F., Mockus, A., Zou, Y., Khomh, F., and Hassan, A. (2013). How does context affect the distribution of software maintainability metrics? In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM)*, pages 350–359.

[177] Zhou, Z.-H. and Liu, X.-Y. (2006). Training cost-sensitive neural networks with methods addressing the class imbalance problem. *IEEE Transactions on Knowledge and Data Engineering*, 18(1):63–77.

# Appendix A

# Terminology and Formalism

The following section introduces the basic *sets* that are used in Chapter 3 and Chapter 4 to define the structural, behavioural and quality features. Then, in Section A.2, the *predicates* used in the feature (*set*) definitions are also presented.

## A.1 Sets

Each object-oriented system consists of a set $C$ of classes, each of which can have different categories of methods. For each class $c \in C$, let:

- $M(c)$ be the set of all methods that are declared (defined) in or inherited by class $c$.
- $M_{INH}(c) \subseteq M(c)$ be the set of methods inherited (and not implemented or overridden) by class $c$.
- $M_{DEC}(c) \subseteq M(c)$ be the set of all methods that are declared inside class $c$.
- $M_{IMM}(c) \subseteq M(c)_{DEC}$ be the set of methods declared inside class $c$ that implement inherited abstract methods.

- $M_{OVR}(c) \subseteq M(c)_{DEC}$ be the set of methods declared inside class $c$ that override inherited concrete (i.e. non-abstract) methods.

- $M_{NEW}(c) \subseteq M(c)_{DEC}$ be the set of methods declared inside class $c$ that do not implement or override any inherited method.

$$\text{where } M_{DEC}(c) = M_{IMM}(c) \cup M_{OVR}(c) \cup M_{NEW}(c)$$
$$\text{and } M_{IMM}(c) \cap M_{OVR}(c) \cap M_{NEW}(c) = \emptyset.$$

Object-oriented classes can also have constructors, which represent a special type of methods that is used to create and instantiate new class instances (objects). The set of constructor methods of each class $c \in C$ is defined as follows:

- $CON(c)$ is the set of all constructor methods that are declared (defined) in class $c$.

While methods and constructor methods can both accept information through *parameters*, only the former can have a *return* value. The sets of parameters and return values in each class $c \in C$ are defined as follows:

- $PAR(m)$ is the set of all parameters that are passed to a method $m$ in class $c$ (i.e. $m \in M_{DEC}(c)$ ) or a set of methods $m$ in class $c$ (i.e. $m \subseteq M_{DEC}(c)$ ). The same applies if $m \in CON(c)$ or $m \subseteq CON(c)$.

- $RET(m)$ is the set of all values that are returned by a method $m$ in class $c$ (i.e. $m \in M_{DEC}(c)$ ) or a set of methods $m$ in class $c$ (i.e. $m \subseteq M_{DEC}(c)$ ).

Besides methods and constructors, classes can also have attributes. For each class $c \in C$, let:

- $A(c)$ be the set of all attributes that are declared in or inherited by class $c$.

- $A_{DEC}(c) \subseteq A(c)$ be the set of all attributes that are declared inside class $c$.

- $A_{INH}(c) \subseteq A(c)$ be the set of all attributes that are inherited by class $c$.

$$\text{where } A(c) = A_{DEC} \cup A_{INH}(c)$$
$$\text{and } A_{DEC} \cap A_{INH}(c) = \emptyset.$$

## A.2 Predicates

The following basic set of predicates have been used in the definitions of the features introduced in this thesis. For each class $c \in C$, attribute $a \in A_{DEC}(c)$ and methods $m$ and $m_2 \in M_{DEC}(c)$ :

- $Private(m)$ is true iff method $m$ is declared as private. The same applies if method $m$ is replaced by attribute $a$.
- $Protected(m)$ is true iff method $m$ is declared as protected. The same applies if method $m$ is replaced by attribute $a$.
- $Public(m)$ is true iff method $m$ is not declared as private or protected. The same applies if method $m$ is replaced by attribute $a$.
- $Abstract(m)$ is true iff method $m$ does not have a body (implementation). Method $m$ can be replaced here by the class $c$ and, in such a case, the predicate is true iff $c$ is declared as *abstract* or if it has one or more abstract methods.
- $Static(m)$ is true iff method $m$ is declared as static, which makes it accessible at the class level as well as the level of instantiated objects. The same applies if method $m$ is replaced by attribute $a$.
- $Final(m)$ is true iff method $m$ is declared as final, which means that its implementation cannot be overridden by subclasses. Method $m$ can be replaced here by attribute $a$ but, in this case, final means that the value initially assigned to $a$ cannot be changed later.

- $SameName(m, m_2)$ is true iff the $m$ and $m_2$ methods have exactly the same name.
- $SameSignature(m, m_2)$ is true iff $SameName(m, m_2)$ predicate is true and also the $m$ and $m_2$ methods have the same set of parameters in the same order.
- $Type(a, c)$ is true iff attribute $a$ has the static class type of class $c$. Attribute $a$ can be replaced here by a parameter $p \in PAR(m)$ or a return type $r \in RET(m)$.
- $Array(a)$ is true iff attribute $a$ is declared as an array, which can include a set of elements of the same type. Attribute $a$ can also be replaced here by a parameter $p \in PAR(m)$ or a return type $r \in RET(m)$.

# Appendix B

# Number of Instances in the Datasets of DP Roles

Table B.1 Number of role-playing classes at different minimum-vote levels.

| Min. Vote | Adapter | | Command | | Composite | | Decorator | | Observer | | Visitor | | Proxy | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Adaptee | Adapter | Receiver | C. Command | Component | Composite | Component | Decorator | Subject | Observer | C. Element | Visitor | RealSubject | Proxy |
| **1** | 61,039 | 44,635 | 25,909 | 26,912 | 4,014 | 7,087 | 3,060 | 9,089 | 14,087 | 7,525 | 2,971 | 1,325 | 633 | 974 |
| | **(20,487)** | **(15,562)** | **(18,517)** | **(19,383)** | **(1,554)** | **(3,072)** | **(1,362)** | **(5,299)** | **(8,141)** | **(4,267)** | **(1,690)** | **(734)** | **(369)** | **(646)** |
| **2** | 14,539 | 12,934 | 6,376 | 6,597 | 591 | 786 | 669 | 884 | 1,684 | 1,442 | 633 | 84 | **113** | **140** |
| **3** | 3,442 | 3,508 | 1,036 | 938 | **174** | **118** | **156** | **113** | **164** | **238** | **417** | **26** | 9 | 18 |
| **4** | **792** | **815** | **120** | **99** | 38 | 24 | 32 | 13 | 1 | 21 | 238 | 11 | 0 | 0 |
| **5** | 111 | 121 | 0 | 0 | 5 | 1 | 3 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| **6** | 3 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

∗Between brackets bold numbers are the number of negative training instances and the other bold numbers are the positive ones.

# Appendix C

# Accuracy Evaluation Table of Phase-Two Classifiers

Table C.1 The test accuracy performance of phase-two SVMs.

| DP | Feature Calculation Method | Precision | | Recall | | F-Measure | |
|---|---|---|---|---|---|---|---|
| | | w/o Q* | w/ Q* | w/o Q* | w/ Q* | w/o Q* | w/ Q* |
| **Adapter** | **Absolute** | 0.025 | 0.028 | 0.051 | 0.051 | 0.034 | 0.036 |
| | **Normalized** | 0.073 | 0.100 | 0.077 | 0.077 | 0.075 | 0.087 |
| **Command** | **Absolute** | 0.800 | 1.000 | 0.308 | 0.308 | 0.444 | 0.471 |
| | **Normalized** | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| **Composite** | **Absolute** | 0.556 | 0.714 | 0.455 | 0.455 | 0.500 | 0.556 |
| | **Normalized** | 0.800 | 0.800 | 0.364 | 0.364 | 0.500 | 0.500 |
| **Decorator** | **Absolute** | 0.500 | 1.000 | 0.250 | 0.250 | 0.333 | 0.400 |
| | **Normalized** | 0.500 | 0.333 | 0.250 | 0.250 | 0.333 | 0.286 |
| **Observer** | **Absolute** | 0.140 | 0.471 | 0.727 | 0.727 | 0.235 | 0.571 |
| | **Normalized** | 0.500 | 0.700 | 0.636 | 0.636 | 0.560 | 0.667 |
| **Visitor** | **Absolute** | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| | **Normalized** | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| **Proxy** | **Absolute** | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| | **Normalized** | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |